sevenstaxTCP/IP
&
sevenstaxRouting

User Manual

Revision No.:	5.03
State:	Release
Author:	sevenstax GmbH

Initial version:	21/08/02
Last change:	23/06/08
Last Review:	03/18/08
Publication:	Public
File name:	sevenstaxTCPIPRout_UserManual_V53

Copyright (c) 2002-2008 by sevenstax GmbH
This document is an intellectual property of sevenstax GmbH. Unauthorized copying and distribution is prohibited.

Table of Contents

1 Preface 7

2 Supported Protocols 9

2.1 Protocols Overview 9
2.2 PPP Support 10
2.3 Ethernet Support 10
2.4 IPv4 Support 10
2.5 Routing Support 10
2.6 TCP Support 10
2.7 UDP Support 10
2.8 DNS Support 11
2.9 ICMP Echo Support 11
2.10 NTP Support 11

3 Internal Data Structures 12

3.1 PACKET_DESC - the Packet Descriptor 12
3.2 IPV4 - the IP Address 12
3.3 CONN_HNDL_TYPE - the Connection Handle (Multi Device Version only) 13

4 Integration into User Application 15

4.1 Overview 15
4.2 Target Settings 16
4.2.1 Alignment and Endianess - features.h 16
4.2.2 Packing - features.h, stx_pack.h, stx_unpack.h 16
4.2.3 System Time - features.h 17
4.2.4 Controlling sevenstaxTCP/IP features - features.h 17
4.2.5 Standard Lib Functions - runtimelib.c/h 18
4.2.6 Protocol spanning Definitions - protdefs.h 18
4.3 Notify Handler - Application Handler Procedure 18
4.3.1 Notify Handler Function Prototype 18
4.3.2 Notify Codes 18
4.3.3 Additional param1 and param2 19
4.3.4 Return Value 19
4.3.5 Registering a Notify Handler 19

4.3.6 Using the Notify Handler..... 19
4.4 Include sevenstax Header Files..... 20

5 IP Module – Single Device..... 22

5.1 Overview..... 22
5.2 Protocol API..... 23
5.2.1 stxIP_Init (handler)..... 23
5.2.2 stxIP_Tick ( )..... 23
5.2.3 stxIP_SetLocalIPAndSubnetMask (psLocalIP, psSubnetMask)..... 23
5.2.4 stxIP_ResetLocalIPAndSubnetMask ( )..... 24
5.2.5 stxIP_GetLocalIP ( )..... 24
5.2.6 stxIP_GetSubnetMask ( )..... 24
5.2.7 stxIP_SetStdGateway (psGateway)..... 24
5.2.8 stxIP_DelStdGateway ( )..... 25
5.2.9 stxIP_GetStdGateway ( )..... 25
5.2.10 stxIP_SaveStdGateway ( )..... 25
5.2.11 stxIP_RestoreStdGateway ( )..... 25
5.2.12 stxIP_AccessLinkLocal (bOn)..... 26
5.3 Notify Codes..... 26
5.4 Interface to the Protocol Layer below..... 27
5.4.1 Accesses from Protocol Layer below into IP Module..... 27
5.4.2 Accesses from IP Module into Protocol Layer below..... 28
5.5 Adjustable Parameters..... 30

6 IP Module – Multi Device..... 31

6.1 Overview..... 31
6.2 Protocol API..... 32
6.2.1 stxIP_Init (handler)..... 32
6.2.2 stxIP_Tick ( )..... 32
6.2.3 stxIP_Attach (hSubConn)..... 32
6.2.4 stxIP_Detach (hConn)..... 33
6.2.5 stxIP_SetLocalIPAndSubnetMask (hConn, psLocalIP, psSubnetMask)..... 33
6.2.6 stxIP_ResetLocalIPAndSubnetMask (hConn)..... 33
6.2.7 stxIP_GetLocalIP (hConn)..... 34
6.2.8 stxIP_GetSubnetMask (hConn)..... 34
6.2.9 stxIP_IsLocalIP (hSubConn, psIPAddress)..... 34
6.2.10 stxIP_SetStdGateway (psGateway, hConn)..... 35
6.2.11 stxIP_DelStdGateway ( )..... 35
6.2.12 stxIP_GetStdGateway (void)..... 35
6.2.13 stxIP_SaveStdGateway ( )..... 36
6.2.14 stxIP_RestoreStdGateway ( )..... 36

6.2.15 stxIP_AccessLinkLocal (hConn, bOn)....36
6.2.16 stxIP_AddRoute (psNetworkTarget, psNetworkMask psGateway, hIPInterface)....37
6.2.17 stxIP_ChangeRoute (uIndex, psNetworkTarget, psNetworkMask, psGateway, hIPInterface)....37
6.2.18 stxIP_DeleteRoute (uIndex)....38
6.2.19 stxIP_IsRouteEmpty (uIndex)....38
6.2.20 stxIP_SetRoutingToLocalSubnet (hConn, eRoutingType)....38
6.2.21 stxIP_SetRoutingToStdGateway (eRoutingType)....39
6.2.22 stxIP_SetRoutingToRoute (uIndex, eRoutingType)....39
6.3 Notify Codes....39
6.4 Interface to the Protocol Layer below....40
6.4.1 Accesses from Protocol Layer below into IP Module....40
6.4.2 Accesses from IP Module into Protocol Layer below....41
6.5 Adjustable Parameters....43

7 Routing....44

7.1 Overview....44
7.2 Forwarding Module....44
7.2.1 Overview....44
7.2.2 Application Programming Interface....44
7.2.2.1 stxForward_Init ( )....44
7.2.2.2 stxForward_Tick ( )....45
7.2.2.3 stxForward_SetFilterFunction (fFilterFunction)....45
7.2.2.4 stxForward_GetSysSecRouted ( )....45
7.2.3 Notify Codes....45
7.2.4 Adjustable Parameters....45
7.3 NAT Module....46
7.3.1 Overview....46
7.3.2 Application Programming Interface....47
7.3.2.1 stxNAT_Init ( )....47
7.3.2.2 stxNAT_Tick ( )....47
7.3.2.3 stxNAT_SetFilterFunction (fFilterFunction)....47
7.3.2.4 stxNAT_SetMasqExtIPFunction (fMasqExtIPFunction)....48
7.3.2.5 stxNAT_GetSysSecRouted ( )....48
7.3.2.6 stxNAT_GetTTL ( )....48
7.3.2.7 stxNAT_SetTTL (bTime)....49
7.3.2.8 stxNAT_AddRoutingTableEntryPerm (psIPInt, uPortInt, uPortMasq, puIndex)....49
7.3.2.9 stxNAT_DelRoutingTableEntryPerm (wIdx)....50

7.3.3 Notify Codes....50
7.3.4 Adjustable Parameters....50

8 TCP Module....51

8.1 Overview....51
8.1.1 TCP Basics....51
8.1.2 Connection Establishment....51
8.1.3 Receiving TCP Packets....52
8.1.4 Transmitting TCP Packets....53
8.1.5 Terminating a TCP Connection....53
8.1.6 Server Functionality....54
8.1.7 Single and Multiple Connections....55
8.2 Protocol API....55
8.2.1 stxTCP_Init (handler)....55
8.2.2 stxTCP_Tick ( )....56
8.2.3 stxTCP_Connect (hSubConn, ip_address, port)....56
8.2.4 stxTCP_Listen (hSubConn, port)....57
8.2.5 stxTCP_WriteBulk (uConnId, data, len, SendDynamic)....57
8.2.6 stxTCP_Disconnect (uConnId, bForced)....58
8.2.7 stxTCP_GetState (uConnId)....59
8.3 Notify Codes....59
8.3.1 NC_TCP_CONNECTED (TCP-Info, ConnID)....60
8.3.2 NC_TCP_CONNTIMEOUT (TCP-Info, ConnID)....60
8.3.3 NC_TCP_FIN (AbortReason, ConnID)....60
8.3.4 NC_TCP_RECV (TCP-Info, ConnID)....60
8.3.5 NC_TCP_TXREADY (TCP-Info, ConnID)....61
8.3.6 NC_TCP_GET_TXDATA (TCP_TxData, ConnID)....61
8.4 Adjustable Parameters....61

9 UDP Module....64

9.1 Overview....64
9.2 Protocol API....64
9.2.1 stxUDP_Init (handler)....64
9.2.2 stxUDP_Tick ( )....64
9.2.3 stxUDP_AddConn (hSubConn, local_port, fProcessFunction)....65
9.2.4 stxUDP_DelConn (uConnId)....65
9.2.5 stxUDP_Write (uConnId, psRemoteIp, uRemotePort, data, datalen)....66
9.3 Notify Codes....66
9.3.1 NC_UDP_RECV (UDP-Info, UDPConnID)....67
9.4 Adjustable Parameters....67

10 DNS Tiny Module 68

10.1 Overview 68

10.2 Protocol API 69

10.2.1 stxDNS_Init (handler) 69

10.2.2 stxDNS_Tick ( ) 69

10.2.3 stxDNS_QueryStart (hConnNetwork, stx dns_ip, www_name) 69

10.2.4 stxDNS_QueryStop (hConnNetwork) 70

10.3 Notify Codes 70

10.3.1 NC_DNS_IP (IP addr, -) 70

10.3.2 NC_DNS_TIMEOUT (- , -) 70

10.4 Adjustable Parameters 71

11 ICMP Module 72

11.1 Overview 72

11.2 Protocol API 72

11.3 Notify Codes 72

11.4 Adjustable Parameters 72

12 NTP Module 73

12.1 Overview 73

12.2 Protocol API 73

12.2.1 stxNTP_Init (handler) 73

12.2.2 stxNTP_Tick ( ) 74

12.2.3 stxNTP_Query (hConnNetwork, ntp_server) 74

12.2.4 stxNTP_Transform (ntp_seconds, seconds, minutes, hours, year, month, day, day_of_week) 75

12.3 Notify Codes 75

12.3.1 NC_NTP_SUCCESS (NTP-sec, -) 75

12.3.1 NC_NTP_FAILURE (- , -) 75

12.4 Adjustable Parameters 75

13 Debugging Support 76

14 Restrictions 77

15 Change History 78

1 Preface

The Internet is today's most commonly used source of information. Almost anywhere in the world (if there is a telephone jack), people using computers can access this vast reservoir of information. Normally, these computers are big, fast and powerful. They have large amounts of memory, processors faster than light and bloated, full-featured operating systems. Because of this computing power, they easily deal with complex network protocols such as PPP and TCP/IP.

In contrast to this, connecting smaller devices (such as embedded systems) to the Internet is somehow a tricky and difficult task. The biggest problem is to put such network protocol software into the limited memory space of small microcontrollers. But is there always a need to do so? The answer is: 'No, sometimes it's enough to implement only the basic parts of a protocol'. This is why sevenstaxTCP/IP was made.

For the reason of smallness, the sevenstaxTCP/IP has no additional protocols (by default) which usually belongs to a complete TCP/IP suite of protocols. For example: sevenstaxTCP/IP cannot be 'pinged' because it doesn't know ICMP (but this can be enabled by conditional compilation). sevenstaxTCP/IP relies on the other end's intelligent control algorithms to improve transmission speed (if it has). Therefore, receiving data with sevenstaxTCP/IP is at an acceptable speed.

To show for what sevenstaxTCP/IP is good for see the following details:

- No receive and transmit buffers. sevenstaxTCP/IP shares buffers with the lower level network protocol (e.g. sevenstaxPPP or sevenstaxEthernet).
- Minimum ROM and RAM usage. Code size differs from 5K to 9K (CISC) depending on the compiler and on the processor on which it is executed. This makes it an excellent choice for small embedded systems.
- Pure TCP/IP. No additional protocols such as ICMP, DHCP, UDP etc. are implemented to save memory. (But some of them can be enabled by conditional compilation - see the chapter 'Currently Supported Add-ons'.)
- Sliding windows support on TCP transmission to maximise send performance.
- Easy integration with existing software. Only a few interfaces.
- Asynchronous behaviour. Other software will not be delayed by sevenstaxTCP/IP.
- Trouble-free operation under an RTOS environments like CMX and embOS.
- Shipped as well-documented source code (ANSI-C) which makes it easy to modify.
- Does not follow RFC in all cases, but fulfils most MUST criteria.
- No dedicated reception buffer

You should give sevenstaxTCP/IP a try when you...

- ...want to bring the Internet to small embedded devices which have small memory sizes, both RAM & ROM.
- ...want to use TCP/IP-communication for systems without TCP/IP stack.
- ...want to build devices which are configurable over a TCP/IP network (printers etc.).

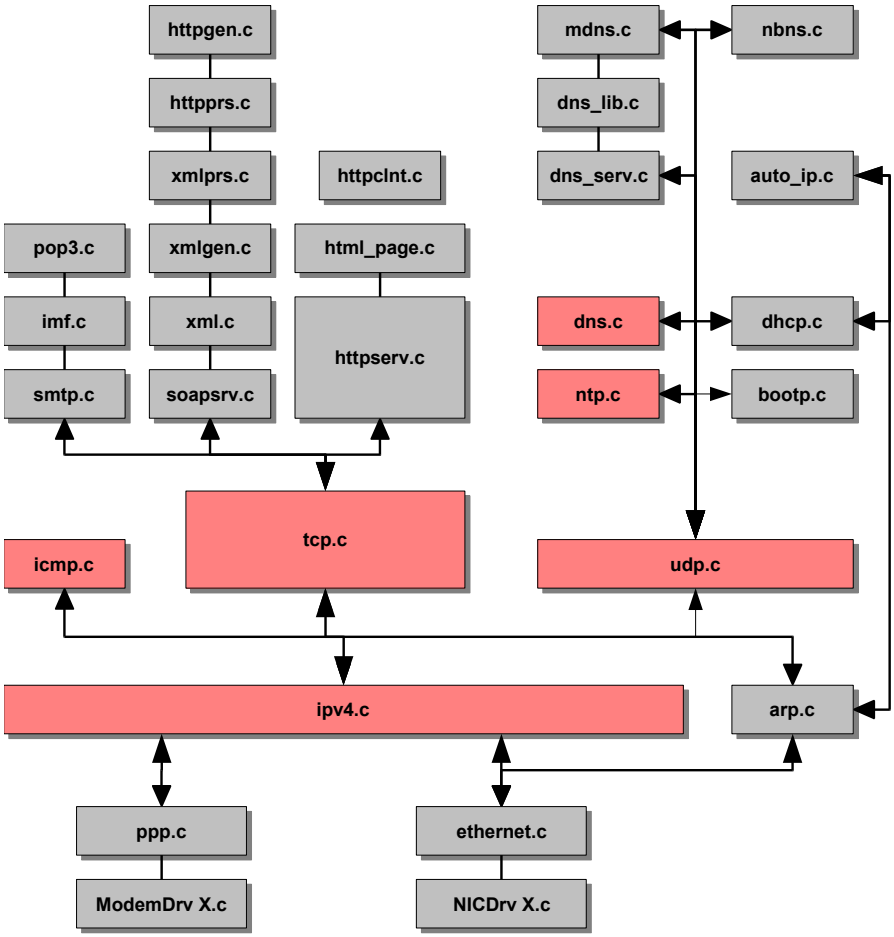
The sevenstaxTCP/IP software stack generally exists in two product versions, called “single device” and “multi device”.
In the smaller single device version, the stack can work with only one connection device like an Ethernet interface or a GSM modem with PPP. On this device you can implement exactly one IP subnet with one IP address.
In the multi device version, the stack can work with multiple connection devices with different IP subnets and IP addresses. You can also define routes to reach remote IP subnets far away. Thus it is possible to act not only as a host but also as a router.

2 Supported Protocols

This manual deals with the main sevenstax internet suite product - sevenstaxTCP/IP. sevenstaxTCP/IP includes the protocols IP, TCP, ICMP, UDP, NTP and a simple DNS client. sevenstaxTCP/IP in this context is used as a general protocol suite name.

2.1 Protocols Overview

The following graphic shows the integration of sevenstaxTCP/IP (red marked) inside the other optional sevenstax internet libraries, available as separate products.



2.2 PPP Support

sevenstaxPPP is available as a separate product.

sevenstaxPPP has been approved in many products and countries worldwide in reliable connections to common public Internet Service Providers (ISPs). sevenstaxTCP/IP connects to PPP, if *PPP_SUPPORTED* = *1* is defined in *features.h*.

2.3 Ethernet Support

sevenstaxEthernet is available as a separate product.

sevenstaxEthernet offers a very tiny and effective protocol implementation for LAN connections, providing implicitly support for ARP and Network Interface Card (NIC) usage. sevenstaxTCP/IP connects to Ethernet, if *ETHERNET_SUPPORTED* = *1* is defined in *features.h*.

2.4 IPv4 Support

IP is part of the sevenstaxTCP/IP product
(files ipv4.c/ipv4md.c, ipv4.h, ipv4defs.h, ipv4lib.c, ipv4lib.h).

IP (Internet Protocol) in version 4 is a core module and can not be disabled. It is available as single device version (ipv4.c) and as multi device version (ipv4md.c).

See section 5 and 6 , IP Module – Single Device and IP Module – Multi Device, for details.

2.5 Routing Support

Routing is part of the sevenstaxRouting product which requires the IP module as multi device version.

Supported operations are NAT (including Port Forwarding) and simple Forwarding (files nat.c, nat.h, natdefs.h, forward.c, forward.h, forwdefs.h).

See section 7, Routing, for details.

2.6 TCP Support

TCP is part of the sevenstaxTCP/IP product (files tcp.c, tcp.h, tcpdefs.h).

TCP is the abbreviation for Transmission Control Protocol, a secure and commonly used packet delivery service. If you need TCP please define TCP_*SUPPORTED* = *1* in *features.h*.

See section 8, TCP Module, for details.

2.7 UDP Support

UDP is part of the sevenstaxTCP/IP product (files udp.c, udp.h, udpdefs.h).

UDP is the abbreviation for User Datagram Protocol, an unreliable packet delivery service. If you need UDP please define *UDP_SUPPORTED* = *1* in *features.h*.

See section 9, UDP Module, for details.

2.8 DNS Support

DNS is part of the sevenstaxTCP/IP product (files dns.c, dns.h, dnsdefs.h).

A simple DNS client can be used by the application to send a query to a DNS server and extract the returned IP address of the response. If you need DNS, please define *DNS_SUPPORTED = 1* in *features.h*.

See section 10, DNS Tiny Module, for details.

If you additionally need DNS server functionality, please use a separate product called *sevenstaxZeroConf*, which includes a DNS server module and a separate DNS library.

2.9 ICMP Echo Support

ICMP is part of the sevenstaxTCP/IP product (files icmp.c, icmp.h, icmpdefs.h).

Sometimes it may be useful that a network host responds to ICMP echo requests (also known as 'PING'). The ICMP support of sevenstaxTCP/IP is very simple and small in code size therefore enabling it will not waste much memory. sevenstaxTCP/IP can be extended to reply to such a request by defining *ICMP_SUPPORTED = 1* in *features.h*.

See section 11, ICMP Module, for details.

2.10 NTP Support

NTP is part of the sevenstaxTCP/IP product (files ntp.c, ntp.h, ntpdefs.h).

A simple NTP client can be used by the application to send a query to a time server and extract the returned time of the response. If you need NTP, please define *NTP_SUPPORTED = 1* in *features.h*.

See section 12, NTP Module, for details.

3 Internal Data Structures

3.1 PACKET_DESC - the Packet Descriptor

```
typedef struct _tag_PACKET_DESC
{
    PACKET_STATUS      state;
    UINT16_stx         size;
    UINT8_stx FPTR_stx data;
    UINT8_stx FPTR_stx payload;
} PACKET_DESC;
```

The packet descriptor is used to hold information about a data packet and for easy passing of this information between internal functions by reference.

PACKET_STATUS state

indicates whether the packet is in use (i.e. under construction) or ready for further processing. It is for internal use only, must not be initialized or changed by the application or lower network components.

FPTR_stx data

points to the location of the first byte in memory. This is the place where the packet's data is stored including link layer headers.

UINT16_stx size

specifies the length in bytes of the entire range pointed to by 'FPTR_stx data'.

FPTR_stx payload

points to the first byte of the IP header. This is usually the first byte after the end of the link layer header.

Packet descriptors are also used by sevenstaxPPP and sevenstaxEthernet to deliver packets to higher network protocols (such as IPv4).

3.2 IPV4 - the IP Address

```
typedef union _tag_IPV4
{
    struct
    {
            UINT8_stx b1;
            UINT8_stx b2;
            UINT8_stx b3;
            UINT8_stx b4;
    } ipb;
    UINT32_stx ipl;
} IPV4;
```

An IP address is actually a 32 bit value which is often shown as 4 decimal values, separated by dots, for better readability (e.g. 192.168.0.51).

IP addresses are stored in network byte order where the first byte (ipb.b1) is the most significant byte. The IPV4 union allows either access to the entire value (ipl) or partly through ipb.b1 to ipb.b4.

Please Note: IP addresses are internally stored as they came over the network (network byte order), regardless of the processor architecture sevenstaxTCP runs on.

3.3 CONN_HNDL_TYPE - the Connection Handle (Multi Device Version only)

```
typedef struct CONN_STRUCT_TYPE
{
    CONN_HNDL_TYPE                              hSubConn;   // connection handle of
                                                            // underlying layer
    CONN_METH_STRUCT_TYPE CONST_stx FPTR_stx    psMeth;     // pointer to function set
    void                            FPTR_stx    psData;     // pointer to data set
    UINT8_stx                       FPTR_stx    pSendBuff;  // send buffer start point
} CONN_STRUCT_TYPE;

typedef struct CONN_STRUCT_TYPE FPTR_stx CONN_HNDL_TYPE;
```

The multi device version of sevenstaxTCP/IP works with so called “connection structures” and “connection handles”. A connection handle is simply a pointer to a connection structure. Based on a connection structure of a layer n protocol you can build a connection on layer n+1 by creating a connection structure in which the entry *hSubConn* points to the connection structure of the protocol on layer n. This connection is done by the application with “attach functions”. The highest layer that works with connections handles is layer 4. Access to layer 4 works with connection id's.

Example:

```
CONN_STRUCT_TYPE sEthernerConnStruct = {...};
CONN_HNDL_TYPE   hConnLayer2         = &sEthernetConnStruct;
CONN_HNDL_TYPE   hConnLayer3         = stxIP_Attach(hConnLayer2);
UINT16_stx       uConnTCP            = stxTCP_Connect(hConnLayer3, &ip, 80);
UINT16_stx       uConnUDP            = stxUDP_AddConn(hConnLayer3, 67, fct);
```

So every protocol knows nothing about the sub layer protocol - it only accesses via *hSubConn* all components of the sub layer connection structure.

Every connection handle holds in *psMeth* a pointer to a structure with function pointers:

```
typedef struct
{
    CONNECTED_FCT_TYPE  fConnected;         // function to return the connected state
    WRITESTR_FCT_TYPE   fWriteString;       // function to send data buffer via sublayer
    WRITEBYTE_FCT_TYPE  fWriteByte;         // function to send single byte via sublayer
    READSTR_FCT_TYPE    fReadString;        // function to read data buffer from sublayer
    READBYTE_FCT_TYPE   fReadByte;          // function to read single byte from sublayer
    SENDBUSY_FCT_TYPE   fSendBusy;          // function to check busy state of sublayer
    GENERIC_FCT_TYPE    fGeneric;           // generic function,
                                            // might be adapted for any purpose
} CONN_METH_STRUCT_TYPE;
```

Not all functions are used in every protocol stack so that some pointers can remain NULL_stx.

The functions are defined as follows:

```
typedef BOOL_stx FCTPTR_stx (CONNECTED_FCT_TYPE) (CONN_HNDL_TYPE              hConn);

typedef BOOL_stx FCTPTR_stx (WRITESTR_FCT_TYPE)  (CONN_HNDL_TYPE              hConn,
                                                  UINT8_stx FPTR_stx          szString,
                                                  UINT16_stx                  uCount);

typedef BOOL_stx FCTPTR_stx (WRITEBYTE_FCT_TYPE) (CONN_HNDL_TYPE              hConn,
                                                  UINT8_stx                   uChar);

typedef BOOL_stx FCTPTR_stx (READSTR_FCT_TYPE)   (CONN_HNDL_TYPE              hConn,
                                                  UINT8_stx FPTR_stx FPTR_stx pszString,
                                                  UINT16_stx FPTR_stx         puCount);

typedef BOOL_stx FCTPTR_stx (READBYTE_FCT_TYPE)  (CONN_HNDL_TYPE              hConn,
                                                  UINT8_stx FPTR_stx          pChar);

typedef BOOL_stx FCTPTR_stx (SENDBUSY_FCT_TYPE)  (CONN_HNDL_TYPE              hConn);

typedef BOOL_stx FCTPTR_stx (GENERIC_FCT_TYPE)   (CONN_HNDL_TYPE              hConn,
                                                  UINT8_stx                   uFctCode,
                                                  void FPTR_stx               pData);
```

Every function has hConn as first parameter. When a function is called, with this parameter a connection structure of the layer, in which this function is defined, is identified.

The *psData* member in the connection structure is a pointer to a data structure. This data structure is specific to the layer so that it is defined as pointer to void.

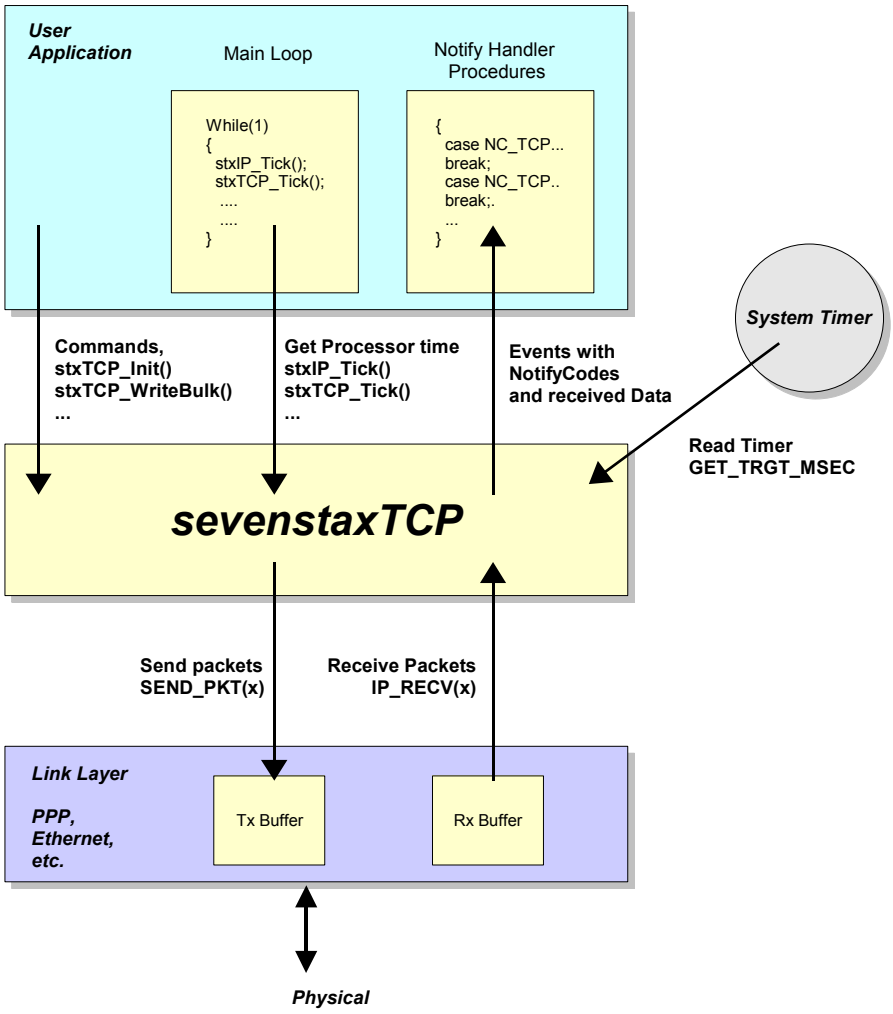
At last *pSendBuff* is a pointer to the send buffer in where the higher protocol can write the data to send.

4 Integration into User Application

4.1 Overview

sevenstaxTCP/IP is designed to work in conjunction with different data link layers such as PPP and/or Ethernet. There are some function prototypes and definitions which must be adjusted according to your application.

Please note: Various functions deal with packet descriptors (see section 3, Internal Data Structures for details). The link layer has to provide the required packet descriptors.



4.2 Target Settings

The main target configuration file is *features.h*. This file is included into all sevenstax modules to control sevenstaxTCP/IP's features.

sevenstaxTCP/IP is implemented to run on every controller, independently of 8/16/32 bit, independently of endianess and compiler behaviour. Every effort has been made to support this universal behaviour on most target systems.

Per default, it is prepared and intended to generate a compiler error of this kind: "*'features.h' not yet adapted to your system! Please check settings in 'features.h'!*".

The following settings need to be checked or changed for your target system.

4.2.1 Alignment and Endianess - features.h

Alignment and endianess are essential settings for a target CPU.
The following settings take effect in the "wordaccess.h".

```
/* target specific settings for byte/struct handling */
/* Please adapt to your target system */
//#pragma BYTEALIGN       /* enable this, if compiler/processor supports byte alignment by
                             default  */
#define BYTEALIGNED     0 /* set to '1', if your target cpu supports byte alignment */
#define LITTLE_ENDIAN   1 /* set to '1', if your target cpu stores multibyte values in
                             network order */
```

4.2.2 Packing - features.h, stx_pack.h, stx_unpack.h

Also packing is an essential setting for a target CPU dealing with multibyte data like IP addresses.

The following defines are placed before and after every structure which needs packing. They can be set globally in the *features.h*:

```
//#define STX_PACK_PRE  /* e.g. set it to '__packed', if ARM compiler used */
//#define STX_PACK_POST /* e.g. set it to '__attribute__((__packed__))', if gcc ...  */
```

Additionally, there are the files stx_pack.h and stx_unpack.h for those compiler directives than can not be mapped in a define, e.g. #pragma pack(1) for packing and #pragma pack(0) for unpacking. They are included before and after the whole structure type definition. You can adapt these files to your application, e.g. if packing should not be disabled for your application.

So, every structure which needs packing is defined as:

```
#include "stx_pack.h"
typedef STX_PACK_PRE struct _tag_EXAMPLE_PACKET
{
   ...
} STX_PACK_POST EXAMPLE_PACKET;
#include "stx_unpack.h"
```

4.2.3 System Time - features.h

sevenstaxTCP/IP needs a system time to execute the protocols timing behaviour. A simple counter delivering milliseconds must be prepared by the user application.

Dedicated macros are prepared to easily integrate the user applications time, e.g. as direct variable access or by a function returning that time. The easiest way is to simply implement a target cpu timer, generating an interrupt every 1 ms and than incrementing the given counters variables below.

```
/* time base for stacks (target system specific) */
/* Please insert your application variables/functions here! */

extern unsigned long    gdwMilliSecCounter;
extern unsigned long    gdwSecCounter;
extern unsigned short   gwMilliSecCounter;
extern unsigned short   gwSecCounter;

#define GET_TRGT_MSEC       gdwMilliSecCounter   /* 32-bit-value, supported e.g. by ISR */
#define GET_TRGT_MSEC_SHORT gwMilliSecCounter    /* 16-bit-value */
#define GET_TRGT_SEC        gdwSecCounter        /* 32-bit-value, supported ISR */
#define GET_TRGT_SEC_SHORT  gwSecCounter         /* 16-bit-value */
```

4.2.4 Controlling sevenstaxTCP/IP features - features.h

This file controls features and therefore the resource requirements of sevenstaxTCP/IP. You can enable/disable a complete protocol support by simply writing a 1 or 0 for the dedicated define. Disabled features won't be compiled and linked. Features that effect only one module are defined in the module specific header file (“[module]defs.h”).

Please do not comment out unused features, use 0 or 1 only.

```
/* protocols */
#define TCP_SUPPORTED           1       /* Enables TCP */
#define UDP_SUPPORTED           1       /* Enables UDP */
#define DNS_SUPPORTED           0       /* Enables simple DNS client */
#define DNS_SRV_SUPPORTED       1       /* Enables enhanced DNS server and client */
#define NTP_SUPPORTED           1       /* Enables NTP (Network Time Protocol) */
#define ICMP_SUPPORTED          1       /* Enables ICMP-Ping responses */
#define HTTPSERV_SUPPORTED      1       /* Enables HTTP server */

/* serial */
#define PPP_SUPPORTED           0       /* Enables PPP */
#define MODEM_SUPPORTED         0       /* Enables Modem driver */
#define NULLMODEM_SUPPORTED     0       /* Enables Nullmodem driver */

/* ethernet */
#define ETHERNET_SUPPORTED      1       /* Enabled Ethernet */
#define MDNS_SUPPORTED          1       /* Enables Multicast-DNS */
#define BOOTP_SUPPORTED         0       /* Enables BOOTP */
#define DHCP_SUPPORTED          1       /* Enables DHCP */
#define AUTOIP_SUPPORTED        1       /* Enables Auto-IP */
#define NBNS_SUPPORTED          1       /* Enables NBNS (NetBIOS Name Service) */
```

Please note that some of the features will only be available if the equivalent sevenstax product was bought.

4.2.5 Standard Lib Functions - runtimelib.c/h

sevenstaxTCP/IP uses some ANSI-C standard library functions for buffer or string manipulation. Several simple embedded compilers do not support a complete implementation of these standard libs. Or your application might use your own memory- or runtime-optimized implementation of the standard lib functions.

To support this, sevenstaxTCP/IP comes with it's own simple runtime lib in “*runtimelib.c*” and “*runtimelib.h*”. Please feel free, to adapt it to your requirements.

4.2.6 Protocol spanning Definitions - protdefs.h

Definitions that are used in multiple protocol stacks are placed here.

In particular these are definitions for the notify handler system, for the interface to the data link layer stacks, for handling single and multi device parameters and for connection structures and handles, that are used in the multi device stack version.

4.3 Notify Handler - Application Handler Procedure

To signal events to the application, sevenstaxTCP/IP needs a handler procedure provided by the application, a so called callback function. Whenever sevenstaxTCP/IP has something to report, it calls this handler procedure.

4.3.1 Notify Handler Function Prototype

The function prototype of this universal notify handler is defined in '*protdefs.h*' and looks like this:

```
typedef UINT32_stx FCTPTR_stx (PROTOCOL_NOTIFY_HANDLER)
       (NOTIFY_CODE code, UINT32_stx param1, UINT32_stx param2);
```

Please note: Check your compiler for declaration format of function pointers. The notify handler needs to be callable from everywhere within the stack!

4.3.2 Notify Codes

The first parameter of the notify handler is the *NOTIFY_CODE*. This is actually an identifier of the message which sevenstaxTCP/IP has sent. The sevenstaxTCP/IP modules use different notify code areas, defined in '*protdefs.h*'.

```
...
#define NC_TCP_MIN                  (NOTIFY_CODE)400
#define NC_TCP_MAX                  (NOTIFY_CODE)419
...
#define NC_UDP_MIN                  (NOTIFY_CODE)420
#define NC_UDP_MAX                  (NOTIFY_CODE)439
...
```

The exact definition of a notify code is located in the export header of the dedicated module, like “tcp.h”:

```
...
#define NC_TCP_CONNECTED            (NOTIFY_CODE)NC_TCP_MIN+0
#define NC_TCP_CONNTIMEOUT          (NOTIFY_CODE)NC_TCP_MIN+1
#define NC_TCP_FIN                  (NOTIFY_CODE)NC_TCP_MIN+2
...
```

You will find a separate list of notify code definitions in the protocol API documentation (e.g. in section 8.3, TCP-Notify Codes).

4.3.3 Additional param1 and param2

The second and third parameters may carry additional information depending on the message. Please see the adequate header file for details.

4.3.4 Return Value

Per default, sevenstaxTCP/IP does not use the return value of the notify handler function of the application. But there are rare cases, in which this value is essential, e.g. in conjunction with *NC_TCP_GET_TXDATA* and *NC_HTTP_DYNINFO*.

For details, please take a closer look into API documentation of the protocols.

4.3.5 Registering a Notify Handler

By calling each *stxXXX_Init()* function with appropriate parameter, the application registers such a handler procedure at sevenstaxTCP/IP.

The application might use always the same handler procedure for all the protocols. In such case, the notify handler might be a very big switch-case construction. Another way is to register dedicated notify handler procedures, so that each handler already has a dedicated responsibility, which might reduce complexity of source code.

In practise, it has been an advantage to distinguish between a common notify handler for all sevenstaxTCP/IP protocols and special notify handlers for modules of other sevenstax products, e.g. sevenstaxWebserver.

4.3.6 Using the Notify Handler

There are some general rules for executing code inside the notify handler procedure:

Leave as fast as possible

When sevenstaxTCP/IP signals events, it cannot proceed before the handler procedure has exited. If you do lengthy tasks inside the handler procedure, the receive buffers may overrun, packets get lost and must be repeated. This will cause considerable delays and might affect performance of the whole system. So be sure not to waste much time here.

Never hold given pointers

The optional given two parameters of the notify handler often are pointers to current connection information and/or user data. They are very connection- and TCP-process- state-specific, so they are valid only as long as the notify handler is being called and active.

After leaving/returning from the notify handler you MUST NOT USE these parameters (pointers) and the referenced data any more, because they will be changed in further TCP Stack internal action.

To use the data referenced by one of the parameters, you might process them immediately (but in very short time) or you might save the content (not the pointer!) into your application.

Never immediately call back into sevenstaxTCP/IP!!!

Because the handler procedure is called from inside sevenstaxTCP/IP, it is dangerous to immediately call back into sevenstaxTCP/IP, by using sevenstaxTCP/IP API functions. Doing so can cause deadlocks or even unlimited recursions which will result in stack overflows and thus crash the system or simply unpredictable reactions and data.

If you like to do some TCP reactions (write/open/close etc.) due to incoming notify codes, execute them AFTER *stxXXX_Tick()* functions. This will prevent any interference between your application and the TCP stack

Do not write to RX/TX buffers

sevenstaxTCP/IP might be in a state where it waits for some data or builds a packet for transmission. Therefore, changing buffer contents can cause sevenstaxTCP/IP to misbehave.

4.4 Include sevenstax Header Files

The sevenstax library code contains a lot of header files (*.h) to define e.g. stack usage, parameters and special prototypes. The sevenstax libraries internally handle all necessary include files for themselves. To reduce complexity, sevenstax decided not to include header files into header files.

The user application might use some or all of the sevenstax internet suite features. To help the developer to find the correct header files and to include them in an order corresponding to the dependencies, sevenstax here gives you a standard dependency list, which might be used in the user application:

```
// please include these common headers in given order
#include "features.h"
#include "stxtypes.h"
#include "protdefs.h"
#include "pktdesc.h"
#include "debugstx.h"

// add protocol specific headers
#include "ipv4.h"
#if (ETHERNET_SUPPORTED==1)
     #include "ethernet.h"
     #include "arp.h"
     #include "nic.h"
#endif
#if (PPP_SUPPORTED==1)
     #include "ppp.h"
#endif
#if (TCP_SUPPORTED==1)
     #include "tcp.h"
#endif
#if (UDP_SUPPORTED==1)
    #include "udp.h"
#endif
#if (HTTPSERV_SUPPORTED == 1)
     #include "httpserv.h"
#endif
#if (DHCP_SUPPORTED==1)
    #include "dhcp.h"
#endif
#if (AUTOIP_SUPPORTED==1)
    #include "auto_ip.h"
#endif
#if (DNS_SUPPORTED == 1) || (DNS_SRV_SUPPORTED == 1)
     #include "dns.h"
#endif
#if (MDNS_SUPPORTED==1)
    #include "mdns.h"
#endif
#if (NBNS_SUPPORTED==1)
    #include "nbns.h"
#endif
#if (NTP_SUPPORTED==1)
    #include "ntp.h"
#endif
....
```

5 IP Module – Single Device

5.1 Overview

Inside sevenstaxTCP/IP the single device version of the IP protocol layer is narrow. A device with this network layer module has exactly one IP address and is part of one subnet. In particular this device can only communicate over one physical interface like Ethernet or a modem. The module holds the own IP address, subnet mask, two DNS servers and a standard gateway.

When receiving packets, the following steps are done:

- ➔ The IP checksum is checked. If the check fails the packet is rejected.
- ➔ Then the target IP address is checked in relation to the current network configuration of the local IP address and the subnet mask.
  If the target IP address is equal to the local IP address the packet is accepted when the local IP address is not 0.0.0.0. In addition the packet is accepted when it is a global/local broadcast or a multicast or. All other packets are rejected.
- ➔ For all unfragmented packets the sub protocol is evaluated. If it is one of the protocols TCP, UDP, ICMP, and if the respective protocol is enabled (XXX_SUPPORTED macro), the packet is delivered to the respective process packet function.

Note:

If target's IP address is part of the own subnet, packets will be sent directly, otherwise they will be sent to the standard gateway. This is done on the underlying data link layer so that the target IP address leaves unchanged and is not the standard gateway IP address.

For some link layer protocols it is necessary to investigate link layer's destination address (e.g. MAC address in Ethernet). Using for instance Ethernet, it might be necessary to send an ARP request to get destination's MAC address. All sevenstax modules use the same send buffer. That means an ARP request has to be sent before building an IP packet.

Because of this, a special “prepare write” function of the link layer has to be defined, which can be called from the IP-module.

This is not necessary in every case. PPP for instance doesn't need this “prepare write“ function”, because of the point to point connection.

5.2 Protocol API

5.2.1 stxIP_Init (handler)

Description

Initializes the IP module. Must be called once before using IP.

Parameter

PROTOCOL_NOTIFY_HANDLER *handler*
Application-defined handler procedure.

Return Value

-

5.2.2 stxIP_Tick ( )

Description

This function keeps IP alive. It must be called as often as possible in order to provide the IP module with processor time.

Parameter

-

Return Value

-

5.2.3 stxIP_SetLocalIPAndSubnetMask (psLocalIP, psSubnetMask)

Description

Sets manually given IP address and subnet mask as local network setting

Parameter

IPV4 FPTR_stx *psLocalIP*
Pointer to local IP address

IPV4 *psSubnetMask*
Pointer to subnet mask of local subnet

Return Value

BOOL_stx

TRUE_stx:	Given IP settings were set successfully
FALSE_stx:	Given IP settings could not be set successfully

5.2.4 stxIP_ResetLocalIPAndSubnetMask ( )

Description

Resets local IP address and subnet mask. Both values are set to 0.0.0.0.

Parameter

-

Return Value

-

5.2.5 stxIP_GetLocalIP ( )

Description

Returns actually set local IP address.

Parameter

-

Return Value

IPV4 FPTR_stx
Pointer to local IP address

5.2.6 stxIP_GetSubnetMask ( )

Description

Returns actually set subnet mask of local subnet.

Parameter

-

Return Value

IPV4 FPTR_stx
Pointer to subnet mask

5.2.7 stxIP_SetStdGateway (psGateway)

Description

Sets manually given IP address as standard gateway

Parameter

IPV4 FPTR_stx *psGatewayIP*
Pointer to IP address of standard gateway

Return Value

BOOL_stx

TRUE_stx:	Standard gateway was set successfully
FALSE_stx:	Standard gateway could not be set successfully

5.2.8 stxIP_DelStdGateway ( )

Description

Deletes the standard gateway

Parameter

-

Return Value

-

5.2.9 stxIP_GetStdGateway ( )

Description

Returns actually set standard gateway

Parameter

-

Return Value

IPV4 FPTR_stx
Pointer to standard gateway

5.2.10 stxIP_SaveStdGateway ( )

Description

Saves actual standard gateway to be restored later.
Can be useful if a standard gateway is set temporarily so that after deletion the old standard gateway can be restored.
See also: stxIP_RestoreStdGateway()

Parameter

-

Return Value

-

5.2.11 stxIP_RestoreStdGateway ( )

Description

Restores saved standard gateway.
Can be useful if a standard gateway is set temporarily so that after deletion the old standard gateway can be restored.
See also: stxIP_SaveStdGateway()

Parameter

-

Return Value

-

5.2.12 stxIP_AccessLinkLocal (bOn)

Description

Controls the access between private and link local IP addresses.

If local IP address is a private IP address, the device can communicate with devices that have a link local IP address (169.254.x.y). If Local IP address is a link local IP address, the device can communicate with devices that have a private IP address.

The module has a request counter so that this function can be called from many modules. If the request counter is at least one the described feature is activated.

Parameter

BOOL_stx *bOn*
If TRUE_stx, the access link local request counter is increased. If FALSE_stx, it is decreased.

Return Value

BOOL_stx
TRUE_stx

5.3 Notify Codes

Although, sevenstaxTCP/IP supports a dedicated notify handler for the IP module, currently no notify codes are defined for IP.

5.4 Interface to the Protocol Layer below

sevenstaxTCP/IP underlying protocols on data link layer might be PPP or Ethernet. The essential interfaces to handle sending and receiving packets via this “sub layer” are implemented as macros in protdefs.h.

5.4.1 Accesses from Protocol Layer below into IP Module

#define LOCAL_IP_ADDRESS sLocalIP

If the local IP address can be set or reset in the data link layer (like in PPP), the data link layer protocol stack can do this directly by accessing this macro.

#define SUBNET_MASK sSubnetMask

If the local subnet mask can be set or reset in the data link layer (like in PPP), the data link layer protocol stack can do this directly by accessing this macro.

#define STD_GATEWAY sGatewayIP

If the standard gateway can be set or reset in the data link layer, the data link layer protocol stack can do this directly by accessing this macro.

#define DNS1_IP_ADDRESS sDNS1IP

If the first DNS server can be set or reset in the data link layer (like in PPP), the data link layer protocol stack can do this directly by accessing this macro.

#define DNS2_IP_ADDRESS sDNS2IP

If the second DNS server can be set or reset in the data link layer (like in PPP), the data link layer protocol stack can do this directly by accessing this macro.

#define IP_RECV(p) stxIP_ProcessPacket(p)

Receive function.

Whenever the link layer has detected an IPv4 packet, it must be delivered to the IP module. Therefore, the link layer has to know the entry point into the IP module where received packets are handled. For this reason, the function *stxIP_ProcessPacket()* is made public by the macro *IP_RECV.* It must be called with a packet descriptor (see section PACKET_DESC - the Packet Descriptor for details) with the following settings:

- state Not used (any value)
- size Size of the entire packet (including link layer headers).
- data A pointer to the first byte of the link layer header.
- payload A pointer to the first byte of the IP header.

The function returns TRUE_stx if the packet was processed and FALSE_stx if not.

5.4.2 Accesses from IP Module into Protocol Layer below

The following macros exist for the sevenstaxEthernet and sevenstaxPPP protocol implementations. Adapt these macros, if you use another sub layer protocol.

#define SEND_PKT(x) stxETH_SendIpFrame(x)
#define SEND_PKT(x) stxPPP_SendIpFrame(x)

One of these functions is the entry point of the link layer, which is responsible for handling data that should be sent off to the network. sevenstaxTCP/IP requires a function which takes a pointer to a packet descriptor as a single parameter. When *SEND_PKT* is called it has done these things before:

- It has build a complete IP packet beginning at the location where the packet descriptor's member '*FPTR_stx payload*' points to.
- It has set the packet descriptor's 'size' member to the size of the IP packet.

The link layer might then do the following:

- Increase the 'size' member about the length of its own headers.
- Build its headers at the location where '*FPTR_stx data*' points.
- Deliver the packet to the network hardware.

From sevenstaxTCP/IP this macro is called with the parameter SENDBUFF.

#define SENDBUFF gETH_sendpacket
#define SENDBUFF gPPP_sendpacket

sevenstaxTCP/IP is designed to reuse the send buffer of the link layer in order to save memory. Therefore SENDBUFF is by default set to one of the above packet descriptors (see section PACKET_DESC - the Packet Descriptor for details) which contain the send buffers.

SENDBUFF must be initialised in the following way:

- The member 'data' must point to first byte of the link layer header. This might be the first byte of the buffer which the link layer will send off to the network.
- The member 'payload' must point to the first byte of the IP header. This should be, for example, fourteen bytes behind 'data' when using sevenstaxTCP/IP on Ethernet.
- '*size*' and '*state*' members are not used at this point and can have any value.

The buffer can be changed according to the projects needs. Therefore the SENDBUFF may be changed to a different packet descriptor but this is normally not required.

#define SEND_BUSY() stxETH_SendBusy()
#define SEND_BUSY() stxPPP_SendBusy()

sevenstaxTCP/IP needs a data link layer function that indicates if this layer is busy with sending a packet. In most cases the data link protocol stack can send a whole packet synchronously inside the call of SEND_PKT() in which the function returns always FALSE_stx (like in sevenstaxEthernet). If a protocol stack needs more time to send a packet it must return TRUE_stx as long as the packet is not sent completely.

#define SEND_PAYLOAD() stxETH_GetSendPayload()
#define SEND_PAYLOAD() stxPPP_GetSendPayload()

With this function macro sevenstaxTCP/IP can ask for the buffer size that is available for the sending payload. In most cases it is the size of the complete send buffer less

the size for the overhead. E.g. for sevenstaxEthernet it is the size of the Ethernet send buffer minus 14 for the Ethernet header size.

This function prevents sevenstaxTCP/IP to write too much data into the send buffer.

#define RCV_PAYLOAD() stxETH_GetRcvPayload()
#define RCV_PAYLOAD() stxPPP_GetRcvPayload()

With this function macro sevenstaxTCP/IP can ask for the buffer size that is available for the received payload. In most cases it is the size of the complete receive buffer less the size for the overhead. E.g. for sevenstaxPPP it is the size of the PPP receive buffer minus 6 for the PPP header and checksum size.

This function have a bearing only on the TCP module, when calculating its window size. So if you don't use TCP, the value returned by the function doesn't matter.

#define RCV_OVERHEAD() stxETH_GetRcvOverhead()
#define RCV_OVERHEAD() stxPPP_GetRcvOverhead()

With this function macro sevenstaxTCP/IP can ask for the memory overhead for saving a received IP packet. In most cases it is the header and footer size of one packet but in some cases it can be more than this, because of special data structures necessary to manage received packets. E.g. for sevenstaxPPP the function returns always 6 (see above), but for sevenstaxEthernet the value is 14 for the header plus 2 because of the ring buffer overhead.

This function have a bearing only on the TCP module, when calculating its optimized window size (see chapter TCP). So if you don't use this feature, the value returned by the function doesn't matter.

#define PREPARE_WRITE(t,d,l) stxETH_PrepareWrite(t,d,l)

If there is a need to prepare the write process you can assign a function to this macro. E.g. for Ethernet it is necessary to send an ARP request before sending the packet out.

t - must contain the type of the destination IP address of the packet to send (see IPV4_PKT_TYPE: unicast, multicast, local broadcast, global broadcast)

d - must point to the destination IP address of the packet to send

l - must point to the local IP address

#define PREPARE_RESPONSE(s,d,l,p) stxETH_PrepareResponse(s,d,l,p)

If there is a need to prepare a response on reception you can assign a function to this macro. E.g. an Ethernet/ARP module can save relevant data in the ARP cache as preparation of a response.

s - must point to the source IP of the received packet

d - must point to the destination IP address of the received packet

l - must point to the local IP address

p - must point to the packet descriptor of the received packet

5.5 Adjustable Parameters

IP_EXTERNAL_BROADCAST_MASK **default: 000000FFh**
valid: 00000000h..FFFFFFFFh

With this macro you can define which local broadcast IP packets, which are not identified because of the actually set subnet mask, are identified as local broadcasts anyway. E.g. if the local subnet mask is set to 255.255.0.0 you would identify IP packets with target IP x.y.z.255 as local broadcasts (which are external because not based on the local subnet mask) if the macro has the default value.

6 IP Module – Multi Device

6.1 Overview

Inside sevenstaxTCP/IP the multi device version of the IP protocol layer is comprehensive. A device with this network layer module can act as a “multi homed host”. This means that it can have more than one IP address and can be part of multiple subnets that are based on one or more physical interfaces like Ethernets or modems (PPP).

The module holds the own IP address and two DNS servers per local subnet.

The main difference compared with the single device version is the existence of a routing table. The own subnets as well as remote subnets are mapped into this table, also the standard gateway. One entry contains

- network target
- network mask
- gateway
- IP interface
- routing type (only with the sevenstaxRouting modules)

With the first two entries, that have IP address format, the (local or remote) subnet is defined. The IP interface defines over which local IP subnet and therefore over which physical device a packet, targeted to a subnet, must be sent. If the gateway entry of one routing table entry has a valid IP address, packets will not be sent directly into the defined (remote) subnet. In this case the packet will be sent on data link layer to this gateway that represents a router. E.g. for Ethernet an ARP request to the gateway is processed before sending the packet.

With the routing type entry the device can not only act as a host but also as a router. At the moment there are the routing types NAT and Forwarding (special modules of sevenstaxRouting). The routing type property can be defined for each entry in the routing table, that means for every defined (local and remote) subnet. This is done with the help of three special API functions of the IP module.

When receiving packets, the following steps are done:

- ➔ The IP checksum is checked. If the check fails the packet is rejected.
- ➔ The local IP subnet over which the packet is received is determined. This is important to check if the target IP is equal to the local IP of this local subnet.
- ➔ Then the target IP address is checked in relation to the current network configuration of the local IP address and the subnet mask.
  If the target IP address is equal to the local IP address the packet is accepted when the local IP address is not 0.0.0.0. In addition the packet is accepted when it is a global/local broadcast or a multicast. All other packets are rejected.
- ➔ For all unfragmented packets the sub protocol is evaluated. If it is one of the protocols TCP, UDP, ICMP, and if the respective protocol is enabled (..._SUPPORTED) the packet is delivered to the respective process packet function.
- ➔ For all packets that are not accepted because of the target IP address or for the packets, where the target IP address is equal to the local IP address but which are

not processed by a higher layer protocol, it is checked if the packet can be routed (only with the sevenstaxRouting modules).

6.2 Protocol API

6.2.1 stxIP_Init (handler)

Description

Initializes the IP module. Must be called once before using IP.

Parameter

PROTOCOL_NOTIFY_HANDLER *handler*
Application-defined handler procedure.

Return Value

-

6.2.2 stxIP_Tick ( )

Description

This function keeps IP alive. It must be called as often as possible in order to provide the IP module with processor time.

Parameter

-

Return Value

-

6.2.3 stxIP_Attach (hSubConn)

Description

Before you can communicate with sevenstaxTCP/IP you have to bind IP to a data link protocol (e.g. Ethernet, PPP). This is done by calling this function in which you specify the data link protocol by a connection handle. Result is a new local IP subnet referenced also by a connection handle.

Parameter

CONN_HNDL_TYPE *hSubConn*
Handle of a data link protocol, must point to a structure of type CONN_STRUCT_TYPE with the configuration described in the chapter “Accesses from IP module into protocol layer below” that you can find below.

Return Value

CONN_HNDL_TYPE
Connection handle of the new IP local subnet. This handle must be used for other API functions that refer to the IP local subnet.

6.2.4 stxIP_Detach (hConn)

Description

Deletes local IP subnet created in stxIP_Attach().

Parameter

CONN_HNDL_TYPE *hConn*
Handle of IP subnet, result of stxIP_Attach() function call

Return Value

-

6.2.5 stxIP_SetLocalIPAndSubnetMask (hConn, psLocalIP, psSubnetMask)

Description

Sets manually given IP address and subnet mask as local network setting for given IP local subnet.

Parameter

CONN_HNDL_TYPE *hConn*
Handle of IP subnet, result of stxIP_Attach() function call

IPV4 FPTR_stx *psLocalIP*
Pointer to local IP address

IPV4 *psSubnetMask*
Pointer to subnet mask of local subnet

Return Value

BOOL_stx

TRUE_stx:	Given IP settings were set successfully
FALSE_stx:	Given IP settings could not be set successfully

6.2.6 stxIP_ResetLocalIPAndSubnetMask (hConn)

Description

Resets local IP address and subnet mask for given IP local subnet. Both values are internally set to 0.0.0.0.

Parameter

CONN_HNDL_TYPE *hConn*
Handle of IP subnet, result of stxIP_Attach() function call

Return Value

-

6.2.7 stxIP_GetLocalIP (hConn)

Description

Returns current local IP address for given IP local subnet.

Parameter

CONN_HNDL_TYPE *hConn*
Handle of IP subnet, result of stxIP_Attach() function call

Return Value

IPV4 FPTR_stx
Pointer to local IP address

6.2.8 stxIP_GetSubnetMask (hConn)

Description

Returns current subnet mask of given IP local subnet.

Parameter

CONN_HNDL_TYPE *hConn*
Handle of IP subnet, result of stxIP_Attach() function call

Return Value

IPV4 FPTR_stx
Pointer to subnet mask

6.2.9 stxIP_IsLocalIP (hSubConn, pslPAddress)

Description

Checks if given IP address is a local IP address for the given data link connection.

Parameter

CONN_HNDL_TYPE *hSubConn*
Handle of data link connection

CONST_stx IPV4 FPTR_stx *pslPAddress*
Pointer to IP address that is to be checked

Return Value

UINT8_stx

0..IP_MULTI-1:	If given IP address is a local IP address
UINT8_stx_MAX:	If given IP address is no local IP address

6.2.10 stxIP_SetStdGateway (psGateway, hConn)

Description

Sets manually given IP address and interface as standard gateway

Parameter

IPV4 FPTR_stx *psGatewayIP*
Pointer to IP address of standard gateway

CONN_HNDL_TYPE *hConn*
Handle of local IP subnet over which the interface is given, result of stxIP_Attach() function call

Return Value

BOOL_stx

TRUE_stx:	Standard gateway was set successfully
FALSE_stx:	Standard gateway could not be set successfully

6.2.11 stxIP_DelStdGateway ( )

Description

Deletes the standard gateway

Parameter

-

Return Value

-

6.2.12 stxIP_GetStdGateway (void)

Description

Returns current standard gateway

Parameter

-

Return Value

IPV4 FPTR_stx
Pointer to standard gateway

6.2.13 stxIP_SaveStdGateway ( )

Description

Saves actual standard gateway to be restored later.
Can be useful if a standard gateway is set temporarily so that after deletion the old standard gateway can be restored.
See also: stxIP_RestoreStdGateway()

Parameter

-

Return Value

-

6.2.14 stxIP_RestoreStdGateway ( )

Description

Restores saved standard gateway.
Can be useful if a standard gateway is set temporarily so that after deletion the old standard gateway can be restored.
See also: stxIP_SaveStdGateway()

Parameter

-

Return Value

-

6.2.15 stxIP_AccessLinkLocal (hConn, bOn)

Description

Controls the access between private and link local IP addresses for a local IP subnet.

If local IP address is a private IP address, the device can communicate with devices that have a link local IP address (169.254.x.y). If Local IP address is a link local IP address, the device can communicate with devices that have a private IP address.

The module has per IP subnet a request counter so that this function can be called from many modules. If the request counter of an subnet is at least one the described feature is activated for this subnet.

Parameter

CONN_HNDL_TYPE *hConn*
Handle of IP subnet, result of stxIP_Attach() function call

BOOL_stx *bOn*
If TRUE_stx, the access link local request counter is increased. If FALSE_stx, it is decreased.

Return Value

BOOL_stx
TRUE_stx

6.2.16 stxIP_AddRoute (psNetworkTarget, psNetworkMask psGateway, hIPInterface)

Description

Adds a route to the routing table..

Parameter

IPV4 FPTR_stx *psNetworkTarget*
IP address space, e.g. 192.168.130.0 If NULL_stx, 0.0.0.0 is inserted

IPV4 FPTR_stx *psNetworkMask*
Network mask, e.g. 255.255.255.0 If NULL_stx, 0.0.0.0 is inserted

IPV4 FPTR_stx *psGateway*
Gateway IP, if hosts in the network are not accessible directly

CONN_HNDL_TYPE *hIPInterface*
Local IP subnet (and so the physical device) over which all packets to the hosts in the given network have to be sent.

Return Value

UINT8_stx
Index of routing table entry – UINT8_stx_MAX if no entry was made.

6.2.17 stxIP_ChangeRoute (uIndex, psNetworkTarget, psNetworkMask, psGateway, hIPInterface)

Description

Changes a route in the routing table.

Parameter

UINT8_stx *uIndex*
Routing table index, result from stxIP_AddRoute()

IPV4 FPTR_stx *psNetworkTarget*
IP address space, e.g. 192.168.130.0, If NULL_stx, 0.0.0.0 is inserted

IPV4 FPTR_stx *psNetworkMask*
Network mask, e.g. 255.255.255.0, If IPV4 NULL_stx, 0.0.0.0 is inserted

IPV4 FPTR_stx *psGateway*
Gateway IP, if hosts in the network are not accessible directly

CONN_HNDL_TYPE *hIPInterface*
Local IP subnet (and so the physical device) over which all packets to the hosts in the given network have to be sent.

Return Value

BOOL_stx
TRUE_stx if route has been changed, otherwise FALSE_stx

6.2.18 stxIP_DeleteRoute (uIndex)

Description

Deletes a route in the routing table.

Parameter

UINT8_stx *uIndex*
Routing table index, result from stxIP_AddRoute()

Return Value

BOOL_stx
TRUE_stx if route has been deleted, otherwise FALSE_stx

6.2.19 stxIP_IsRouteEmpty (uIndex)

Description

Checks if the denoted routing table entry is empty.

Parameter

UINT8_stx *uIndex*
Routing table index, result from stxIP_AddRoute()

Return Value

BOOL_stx
TRUE_stx if route is empty, otherwise FALSE_stx

6.2.20 stxIP_SetRoutingToLocalSubnet (hConn, eRoutingType)

Description

Sets the routing property to the given local IP subnet. When activated all packets from another IP subnet that are directed to the given IP subnet are routed. The default setting is “no routing” in which case all these packets are rejected.

Parameter

CONN_HNDL_TYPE *hConn*
Handle of local IP subnet, result of stxIP_Attach() function call

IPV4_ROUTING_TYPE *eRoutingType*
Routing type. In this version, there are the following possibilities:
0 – IPV4_NO_ROUTING: no routing
1 – IPV4_ROUTING_FORWARD: simple forwarding
2 – IPV4_ROUTING_NAT: network address translation

Return Value

BOOL_stx
TRUE_stx, if routing property was set
FALSE_stx, if routing property could not be set

6.2.21 stxIP_SetRoutingToStdGateway (eRoutingType)

Description

Sets the routing property to the IP subnet that is located behind the standard gateway. When activated all packets from another IP subnet that are directed to that subnet are routed. The default setting is “no routing” in which case all these packets are rejected. This function can for example enable one or more another IP subnets to access the internet.

Parameter

IPV4_ROUTING_TYPE *eRoutingType*
Routing type. In this version there are the following possibilities:
0 – IPV4_NO_ROUTING: no routing
1 – IPV4_ROUTING_FORWARD: simple forwarding
2 – IPV4_ROUTING_NAT: network address translation

Return Value

BOOL_stx
TRUE_stx

6.2.22 stxIP_SetRoutingToRoute (uIndex, eRoutingType)

Description

Sets the routing property to an arbitrary local or remote IP subnet (except for the subnet behind the standard gateway), defined by the index of the routing table. As index the result of the call of the function *stxIP_AddRoute()* must be used. When activated, all packets from another IP subnet that are directed to that subnet are routed. The default setting is “no routing” in which case all these packets are rejected.

Parameter

UINT8_stx *uIndex*
Routing table index, result from stxIP_AddRoute()

IPV4_ROUTING_TYPE *eRoutingType*
Routing type. In this version there are the following possibilities:
0 – IPV4_NO_ROUTING: no routing
1 – IPV4_ROUTING_FORWARD: simple forwarding
2 – IPV4_ROUTING_NAT: network address translation

Return Value

BOOL_stx
TRUE_stx, if routing property was set
FALSE_stx, if routing property could not be set

6.3 Notify Codes

Although, sevenstaxTCP/IP supports a dedicated notify handler for the IP module, currently no notify codes are defined for IP.

6.4 Interface to the Protocol Layer below

sevenstaxTCP/IP underlying protocols might be PPP and/or Ethernet and/or any other data link protocol.

The essential interfaces to handle sending and receiving packets via this “sub layer” are described here.

Feel free to adapt this interface to your application, if you use another sub layer protocol.

6.4.1 Accesses from Protocol Layer below into IP Module

#define IP_SET_LOCAL_IP_AND_SUBNET_MASK_LINK(hSubConn, plp, pSubnet)
stxIP_SetLocalIPAndSubnetMask_Link(hSubConn, plp, pSubnet)

If the local IP address and local subnet mask can be set in the data link layer (like in PPP), the data link layer protocol stack can do this by accessing this macro.
The first parameter is the connection handle of the data link layer connection.

#define IP_RESET_LOCAL_IP_AND_SUBNET_MASK_LINK(hSubConn)
stxIP_ResetLocalIPAndSubnetMask_Link(hSubConn)

If the local IP address and local subnet mask can be reset in the data link layer (like in PPP), the data link layer protocol stack can do this by accessing this macro.
The first parameter is the connection handle of the data link layer connection.

#define IP_SET_DNS1_IP_LINK(hSubConn, plp)
stxIP_SetDNS1IP_Link(hSubConn, plp)

If the first DNS server can be set in the data link layer (like in PPP), the data link layer protocol stack can do this directly by accessing this macro.
The first parameter is the connection handle of the data link layer connection.

#define IP_RESET_DNS1_IP_LINK(hSubConn)
stxIP_ResetDNS1IP_Link(hSubConn)

If the first DNS server can be reset in the data link layer (like in PPP), the data link layer protocol stack can do this directly by accessing this macro.
The first parameter is the connection handle of the data link layer connection.

#define IP_SET_DNS2_IP_LINK(hSubConn, plp)
stxIP_SetDNS2IP_Link(hSubConn, plp)

If the second DNS server can be set in the data link layer (like in PPP), the data link layer protocol stack can do this directly by accessing this macro.
The first parameter is the connection handle of the data link layer connection.

#define IP_RESET_DNS2_IP_LINK(hSubConn)
stxIP_ResetDNS2IP_Link(hSubConn)

If the second DNS server can be reset in the data link layer (like in PPP), the data link layer protocol stack can do this directly by accessing this macro.
The first parameter is the connection handle of the data link layer connection.

#define IP_RECV(h,p) **stxIP_ProcessPacket(h,p)**

Receive function.

Whenever the link layer has detected an IPv4 packet, it must be delivered to the IP module. Therefore, the link layer has to know the entry point into the IP module where received packets are handled. For this reason the function *stxIP_ProcessPacket()* is made public by the macro *IP_RECV*. It must be called with the connection handle of

the data link connection and with a packet descriptor (see section PACKET_DESC - the Packet Descriptor for details) with the following settings:

- state Not used (any value)
- size Size of the entire packet (including link layer headers).
- data A pointer to the first byte of the link layer header.
- payload A pointer to the first byte of the IP header.

The functions returns TRUE_stx if the packet was processed and FALSE_stx if not.

6.4.2 Accesses from IP Module into Protocol Layer below

All accesses into the protocol layer below (data link layer) are done via the connection handle of this protocol layer which was connected to the IP module with the function *stxIP_Attach()*.

This connection handle (CONN_HNDL_TYPE) is a pointer to a connection structure (CONN_STRUCT_TYPE). To work properly this structure must have the following configuration:

- ***hSubConn***:
  Don't care, any value, isn't accessed from IP module
- ***psMeth***:
  Must point to a structure of type CONN_METH_STRUCT_TYPE with the following contents:
  - *fConnected*:
    Must point to a function of type CONNECTED_FCT_TYPE, that indicates if a connection over the given connection handle is active.
  - *fWriteString*:
    Must point to a function of type WRITESTR_FCT_TYPE, over which IP packets can be sent.
  - *fWriteByte*:
    Don't care, any value, isn't accessed from IP module
  - *fReadString*:
    Don't care, any value, isn't accessed from IP module
  - *fReadByte*:
    Don't care, any value, isn't accessed from IP module
  - *fSendBusy*:
    Must point to a function of type SENDBUSY_FCT_TYPE, that indicates if the protocol stack has still some data to send
  - *fGeneric*:
    If there is a need to prepare the write process or to prepare a response on reception, this entry must point to a function of type GENERIC_FCT_TYPE. E.g. for Ethernet it is necessary to send an ARP request before sending the packet out. Also an Ethernet module can save relevant data in the ARP cache as preparation of a response.

Preparing a write process:
The generic function is called with function code (*uFctCode*) L2_GENERIC_FCT_PREPARE_WRITE_IDX.
The data parameter (*pData*) points to a structure of type L2_GENERIC_FCT_PREPARE_WRITE_PARA_STRUCT with following entries belonging to the packet to send:
uIPPktType: IP packet type (enum IPV4_PKT_TYPE)
psDestIP: pointer to destination IP address
psLocalIP: pointer to local IP address

Preparing a response:
The generic function is called with function code (*uFctCode*) L2_GENERIC_FCT_PREPARE_RESPONSE_IDX.
The data parameter (*pData*) points to a structure of type L2_GENERIC_FCT_PREPARE_RESPONSE_ PARA_STRUCT
with following entries belonging to the received packet:
psSrcIP: pointer to source IP address
psDestIP: pointer to destination IP address
psLocalIP: pointer to local IP address
pPkt: pointer to packet descriptor

- ***psData***:
  Must point to a structure of type CONN_DATA_L2_STRUCT_TYPE with the following contents:
  - *uSizeSendPayload*:
    Buffer size that is available for the sending payload. In most cases it is the size of the complete send buffer less the size for the overhead. E.g. for Ethernet it is the size of the Ethernet send buffer minus 14 for the Ethernet header size. This value prevents sevenstaxTCP/IP to write too much data into the send buffer.
  - *uSizeRcvPayload*:
    Buffer size that is available for the received payload. In most cases it is the size of the complete receive buffer less the size for the overhead. E.g. for PPP it is the size of the PPP receive buffer minus 6 for the PPP header and checksum size.
    Note: This value has a bearing only on the TCP module, when calculating its window size depending on this value. So if you don't use TCP, the value doesn't matter.
  - *uSizeRcvOverhead*:
    Memory overhead for saving a received IP packet. In most cases it is the header and footer size of one packet but in some cases it can be more than this because of special data structures necessary to manage received packets. E.g. for sevenstaxPPP the function returns always 6 (s.a.), but for sevenstaxEthernet the value is 14 for the header plus 2 because of the ring buffer overhead.
    Note: This value has a bearing only on the TCP module, when calculating its optimized window size depending on this value (see chapter TCP). So if you

don't use TCP or this feature, the value returned by the function doesn't matter.

- ***pSendBuff***:
  Must point into that position in the send buff, where a sending IP packet can be placed.
  Note: To reduce copying, it would be effective to leave space for the eventually existing header of the data link protocol. E.g. for sevenstaxEthernet this pointer points to the beginning of the send buffer plus 14.

6.5 Adjustable Parameters

IP_EXTERNAL_BROADCAST_MASK **default: 000000FFh**
valid: 00000000h..FFFFFFFFh

With this macro you can define which local broadcast IP packets, not been identified because of the actually set subnet mask, are identified as local broadcasts anyway. E.g. if the local subnet mask is set to 255.255.0.0 you would identify IP packets with target IP x.y.z.255 as local broadcasts (which are external because not based on the local subnet mask) if the macro has the default value.

IP_MULTI **default: 1**
valid: 1..255

With this macro you can define the maximum number of local IP subnets. This value corresponds to the number of successful calls of stxIP_Attach().

IP_CNT_ROUTING_TAB_ENTRIES **default: 10**
valid: 1..255

With this macro you can define the number of entries in the routing table. (Internally the routing table has one more entry which is a special entry for the standard gateway).

7 Routing

7.1 Overview

The normal operation mode of sevenstaxTCP/IP is acting as a host, that means communication endpoint, in the network. With help of the sevenstaxRouting modules you can implement a network in which the device can have an intermediary property and so acting as a router.

The actual routing modules work on the IP layer. They contain the routing mechanism NAT (network address translation), implemented as NAPT (network address port translation), and Forwarding. The enabling of these routing features can be done per (local or remote) IP target subnet in the Multi-Device-IP module with special API functions. The translations of the packets are done in the routing modules called from the Multi-Device-IP module.

7.2 Forwarding Module

7.2.1 Overview

With the forwarding module a packet can be forwarded from one to another IP subnet. It is used when the sender shall remain directly addressable. The IP packet leaves unchanged except for the TTL field (time to live) that is decreased by one. Because of this little change the IP checksum is recalculated for every new packet.

A requirement for successfully routing a packet is that the receive buffer size of the source IP subnet is equal to the send buffer size of the destination IP subnet.

The module can be configured so that for TCP and UDP packets the checksum is checked and all packets where the check failed are rejected.

In addition the user can define a filter function to filter out some packets with certain properties concerning destination IP address, transport layer protocol (e.g. UDP) or destination port of transport layer protocol.

This module can be activated by defining FORWARD_SUPPORTED with 1 (*features.h*).

7.2.2 Application Programming Interface

7.2.2.1 stxForward_Init ( )

Description

Initializes the Forwarding module. Must be called once before using Forwarding.

Parameter

-

Return Value

-

7.2.2.2 stxForward_Tick ( )

Description

This function keeps Forwarding alive. It must be called as often as possible in order to provide the Forwarding module with processor time.

In this version this function does nothing.

Parameter

-

Return Value

-

7.2.2.3 stxForward_SetFilterFunction (fFilterFunction)

Description

Sets a filter function which can filter out some packets with certain properties concerning destination IP address, transport layer protocol (e.g. UDP) or destination port of transport layer protocol.

Parameter

FW_FILTER_FUNCTION *fFilterFunction*
Filter function

Return Value

-

7.2.2.4 stxForward_GetSysSecRouted ( )

Description

Returns the second counter time stamp of the last routed IP packet.

Parameter

-

Return Value

UINT16_stx
Second counter time stamp of last routed IP packet

7.2.3 Notify Codes

Currently no notify codes are defined for the Forwarding module.

7.2.4 Adjustable Parameters

FW_CHECK_CHECKSUM **default: 1**
valid: 0,1

With this parameter you can enable the checking of the checksum of delivered TCP and UDP packets.

7.3 NAT Module

7.3.1 Overview

With the NAT module a packet can be routed from one to another IP subnet. It is used when the sender is not addressable from the target IP subnet. In most cases it is used when the packet has to be routed into the Internet.

When routing into the target IP subnet, the internal source IP address must be translated into an external IP address that is reachable from the target IP's subnet (“Source NAT”). In case of routing in the other direction, the external destination IP address must be translated into the internal IP address (“Destination NAT”). In addition the existing internal send port of the transport layer protocol (e.g. TCP, UDP) is translated into another (masqueraded) port to assign the returned packets to the internal sender. Also the TTL field (time to live) is decreased by one. Because of this change the IP checksum is recalculated for every routed packet.

The translation information is stored in a translation table. Every entry contains:

- the internal IP address
- the internal transport layer port
- the masqueraded transport layer port
- time stamp (second counter) of the time when the entry was added or used

One entry is added automatically when a packet is routed into the target IP subnet. When an entry has not been used for a given time it is marked as rewritable. As long as this entry has not been rewritten, it can be reactivated by an appropriate packet with the old values that are still located there.

An entry can also be inserted manually to activate a permanent translation. This operation is called “port forwarding”. These entries are never rewritten automatically and must be deleted manually.

A requirement for successfully routing a packet is that the receive buffer size of the source IP subnet is equal to the send buffer size of the destination IP subnet.

The module can be configured so that for TCP and UDP packets the checksum is checked and all packets where the check failed are rejected.

In addition the user can define a filter function to filter out some packets with certain properties concerning destination IP address, transport layer protocol (e.g. UDP) or destination port of transport layer protocol.

As described above in normal operation only the internal IP address is exchanged during the routing process. By registering a special function the external IP address (target IP address of packet directed to target IP subnet) can additionally be exchanged (masqueraded). This transformation is called “Twice-NAT” because both IP addresses are exchanged in every routed packet. This can be useful for example to reach IP addresses that normally would not be routed as a result of the IP routing table.

This module can be activated by defining NATPF_SUPPORTED with 1 (*features.h*).

7.3.2 Application Programming Interface

7.3.2.1 stxNAT_Init ( )

Description

Initializes the NAT module. Must be called once before using NAT.

Parameter

-

Return Value

-

7.3.2.2 stxNAT_Tick ( )

Description

This function keeps NAT alive. It must be called as often as possible in order to provide the NAT module with processor time.

Parameter

-

Return Value

-

7.3.2.3 stxNAT_SetFilterFunction (fFilterFunction)

Description

Sets a filter function which can filter out some packets with certain properties concerning destination IP address, transport layer protocol (e.g. UDP) or destination port of transport layer protocol.

Parameter

NAT_FILTER_FUNCTION *fFilterFunction*
Filter function

Return Value

-

7.3.2.4 stxNAT_SetMasqExtIPFunction (fMasqExtIPFunction)

Description

Sets a masquerading function which can exchange the external IP address. When such a function is set, both IP addresses are translated for every routed packet (“Twice-NAT”).
This feature only is active if NAT_MASQUERADE_EXT_IP_ADDR is set to 1.

Parameter

NAT_MASQ_EXT_IP_FUNCTION *fMasqExtIPFunction*
Masquerading function

Return Value

-

7.3.2.5 stxNAT_GetSysSecRouted ( )

Description

Returns the second counter time stamp of the last routed IP packet.

Parameter

-

Return Value

UINT16_stx
Second counter time stamp of last routed IP packet

7.3.2.6 stxNAT_GetTTL ( )

Description

Returns actual time to live value for an entry in the translation table.
This function only is available if NAT_ENABLED == 1.

Parameter

-

Return Value

UINT8_stx
Actual time to live value for an entry in the translation table.

7.3.2.7 stxNAT_SetTTL (bTime)

Description

Sets time to live value for an entry in the translation table.
This function only is available if NAT_ENABLED == 1.

Parameter

UINT8_stx *bTime*
new time to live value

Return Value

NAT_ERR
NAT_ERR_OK if value was set successfully
NAT_ERR_INVALID_PARAM, if value could not be set successfully because it was zero

7.3.2.8 stxNAT_AddRoutingTableEntryPerm (pslPInt, uPortInt, uPortMasq, puIndex)

Description

Inserts a permanent entry in the translation table (“port forwarding”).
This function only is available if PFW_ENABLED == 1.

Parameter

CONST_stx IPV4 FPTR_stx *pslPInt*
private IP address

CONST_stx UINT16_stx *uPortInt*
private transport layer port

CONST_stx UINT16_stx *uPortMasq*
masqueraded transport layer port

UINT8_stx FPTR_stx *puIndex*
index of translation table, UINT8_stx_MAX, if no free entry was found

Return Value

NAT_ERR

NAT_ERR_OK,	on successful operation
NAT_ERR_NOT_FOUND,	no free entry found
NAT_ERR_FAILURE,	masq port is already used for another entry

7.3.2.9 stxNAT_DelRoutingTableEntryPerm (wIdx)

Description

Deletes a permanent entry (“port forwarding”) in the translation table.
This function only is available if PFW_ENABLED == 1.

Parameter

UINT8_stx *wIdx*
index of translation table, result of *stxNAT_AddRoutingTableEntryPerm()*

Return Value

NAT_ERR

NAT_ERR_OK,	if permanent entry was deleted successfully
NAT_ERR_INVALID_PARAM,	if permanent entry could not be deleted successfully because the index had a wrong value

7.3.3 Notify Codes

Currently no notify codes are defined for the Forwarding module.

7.3.4 Adjustable Parameters

NAT_ENABLED **default: 1**
valid: 0,1

With this parameter you can enable the normal NAT operation:
Routing initiated from IP subnet not equal to target IP subnet,
temporary translation table entries.

PFW_ENABLED **default: 0**
valid: 0,1

With this parameter you can enable the port forwarding:
Routing initiated from target IP subnet or not target IP subnets,
permanent translation table entries.

NAT_STD_TTL **default: 5 (30 sec.)**
valid: 1..255 (6..1530 sec.)

Default TTL (time to live) value for an entry in the translation table (in 1/10 min.)

NAT_CNT_ROUTING_TAB_ENTRIES **default: 40**
valid: 1..255

Count of entries in the translation table.

NAT_CHECK_CHECKSUM **default: 1**
valid: 0,1

With this parameter you can enable the checking of the checksum of delivered TCP and UDP packets.

NAT_MASQUERADE_EXT_IP_ADDR **default: 1**
valid: 0,1

With this parameter you can enable the feature to masquerade the external IP address (see also: stxNAT_ SetMasqExtIPFunction()).

8 TCP Module

8.1 Overview

TCP is part of sevenstaxTCP/IP standard Internet library. It can be enabled by setting *TCP_SUPPORTED* to 1 in *features.h* (see section 2, Supported Protocols).

8.1.1 TCP Basics

TCP is a connection-oriented protocol. This means that TCP has to compensate these deranging side effects of the underlying network:

- Packet loss
- Packet duplication
- Out-of-order data delivery

To perform these tasks, some kind of acknowledging and confirmation must be used. For this purpose, TCP packets carry additional information such as sequence/acknowledge numbers and bit flags. Before two Hosts using TCP can exchange data, they must synchronize (connect) to each other. This is done by both stations, exchanging a special sequence of datagrams where they tell their initial sequence numbers. During operation, each TCP knows what the other end awaits and tries to deliver the packets that the other TCP has 'ordered'. Also it manages a buffer with already transmitted data to repeat transmission if the recipient hasn't acknowledged the packets. After both Hosts have finished, the connection must be closed. This can be done either by exchanging datagrams similar to the connection process or (faster) by sending a 'connection reset' from any station. Of course, this is a very much incomplete description of TCP - a TCP/IP tutorial is beyond the scope of this document.

8.1.2 Connection Establishment

For the purpose of connection establishment, sevenstaxTCP has two interfaces named *stxTCP_Connect()* (client mode) and *stxTCP_Listen()* (server mode) (how a connection is established is shown in RFC 793, figure 7).

NOTE: For *stxTCP_Connect()*, the local port numbers of sevenstaxTCP are pseudo-random values.

sevenstaxTCP tries to send some SYNs to the other end until it receives a SYNACK. If, for any reason, no SYNACK arrives, sevenstaxTCP notifies the application (through the handler procedure described below) that a connection isn't possible.

When sevenstaxTCP runs as a TCP server (like a web server), it is also able to respond to connection attempts issued from remote TCPs. Instead of calling *stxTCP_Connect()*, the application should call *stxTCP_Listen()* to bring sevenstaxTCP into the listen state. For further information refer to the chapter 'Currently Supported Add-ons' (Server functionality).

Also sevenstaxTCP has a built-in mechanism to handle half-open connections in server mode. When a connection is established and the remote client reconnects (e.g. after a possible system crash) with the same local port number, sevenstaxTCP

sends a RST (TCP reset) to the client. The RST cancels the old connection and a subsequent connection request will succeed.

8.1.3 Receiving TCP Packets

If the lower network layer (in most cases an IPv4 implementation, e.g. sevenstaxIP) has recognized a TCP packet, it calls sevenstaxTCP's receive function *stxTCP_ProcessPacket()*. Before calling this function, the lower network layer prepares a packet descriptor (an internal data structure described below) which holds information about the TCP packet.

Usually, the packet resides in the receive buffer of the lower network layer thus sevenstaxTCP doesn't need a buffer itself. The *stxTCP_ProcessPacket()* function does everything necessary on reception including data delivery to the application, handling connection requests and transmitting response packets back to the other end.

Processing Received TCP Packets

1. In “single connection mode” :
   The destination port is compared to the local port. If they are not equal the packet is rejected.
2. In “single connection mode” and multi device version:
   If the sub connection handle is not NULL_stx and not equal to the locally set sub connection handle the packet is rejected.
3. If a SYN was received:
   - A listening instance is searched (in “multi connection mode”)
   - If found, the checksum is checked (if enabled)
   - If check was successful, all relevant data and status is saved
   Exit
4. In “multi connection state”:
   An instance is searched where the remote and local ports matches.
   In multi device version the sub connection handle must also match.
5. In “single connection state”:
   The remote port is compared to the locally set remote port. If they are not equal the packet is rejected.
6. The checksum is checked (if enabled)
7. The remote IP address is compared to the locally set remote IP address. If they are not equal the packet is rejected.
8. If a SYNACK was received (after a SYN has been sent), an ACK will be returned, the alive timer initialized, the first acknowledge number calculated and the application notified that the connection is established and data may be transferred. At this point no further packet will be handled.
9. If a RST was received (e.g. the server has detected a TCP error) the connection will be aborted and the application notified of this event. No further packet handling will be done.
10. If a connection is established, the received sequence number will be compared with the expected one. This means we have received a packet in the right order which will then be delivered to the upper layer (if it carries

data). Also the expected sequence number for the next packet will be calculated here.

11. If a connection is established, and the received sequence number does not match the expected one, the own acknowledge number will be set to the expected sequence number and an ACK segment with this number is sent to the peer to force a retransmission. At this point, the handling of this packet will be aborted.
12. If a FINACK is received during an open connection (the peer wants to close the connection) an ACK and (later) a FINACK is returned and the connection is aborted. Furthermore the application will be notified of the connection's termination (after the FINACK). No further packet handling will be executed.
13. If a FINACK is received after sent FINACK and received ACK (we wanted to close the connection) an ACK is returned and the connection is aborted. Furthermore the application will be notified of the connection's termination. No further packet handling will be executed.
14. In case an ACK is received while a connection is established, a data packet must be acknowledged. If the payload of the packet is not empty the packet is delivered to the application via notify code NC_TCP_RECV.
15. In case an ACK is received while no connection is established this may indicate that a half-open connection exists. sevenstaxTCP will send an RST which will close the old connection enabling a new one to be established on the same port.

8.1.4 Transmitting TCP Packets

To transmit TCP packets, sevenstaxTCP needs to know two things from the lower network layer. First it needs the address of the transmit buffer, because sevenstaxTCP must know where to assemble a TCP packet. Next, sevenstaxTCP needs to know the transmit function of the lower layer. This function takes the TCP packet and forwards the packet to the network. In single device version this function might be an Ethernet or PPP send function, thus a function on data link layer. In multi device version a send function of the network layer (IP layer) is called via the sub connection handle.

Immediately after sevenstaxTCP has built the packet in memory, it prepares the packet descriptor (in single device version) and calls the transmit function to send out the packet.

Inside sevenstaxTCP there is a function named *stxTCP_InternalSend()* which performs all of the above described tasks.

8.1.5 Terminating a TCP Connection

Usually, a web server terminates the TCP connection after it has transferred the last byte of the page. When the server wants to close the connection, it sends a FINACK and enters the FIN_WAIT_1 state (see RFC 793, figure 13).

Users of sevenstaxTCP must call the function *stxTCP_Disconnect()* which initiates the closing process. When the connection is closed, sevenstaxTCP calls the application's handler procedure by posting a NC_TCP_FIN event.

There are some other conditions, when sevenstaxTCP issues NC_TCP_FIN as well. For example: If sevenstaxTCP sends data (e.g. an HTTP request) but the packet was not acknowledged after a few retries (ACK timeout), the TCP connection will also be

terminated. To separate regular terminations from error conditions, the handler procedure is called with different parameters. For a detailed description see the following table (the constants are defined in tcp.h, enum ABORT_REASON):

Cause of termination	***First parameter of handler procedure (ABORT_REASON)***	***Meaning***
A successful call to stxTCP_Disconnect() or the remote TCP has closed the connection.	AR_CLOSED	Normal operation. All data has been successfully transferred.
1. stxTCP_Disconnect() was called and sevenstaxTCP was in transition state (neither in the connected nor in the closed state). 2. The other end has send an RST. 3. Timeout. The remote TCP does not respond to our FINACK's. or FIN's	AR_CLOSE_FORCED	Error. Some data might be received.
Transmitted data was not acknowledged after some retransmissions.	AR_RTM_EXCEEDED	Error. Too many retransmissions.
ARP was not able to resolve the MAC address of the other side. (Ethernet only)	AR_NO_MACADDRESS	Error. TCP can not continue. Maybe peer does not exist on the LAN.
Connection over network layer has terminated.	AR_NO_NETWORK	Error. TCP can not continue.

8.1.6 Server Functionality

sevenstaxTCP is capable to act as a server. This means it can passively wait for connections from other Hosts and send requested data or error messages to them. To enable the server functionality the following steps are required:

- After initialization of sevenstaxTCP call *stxTCP_Listen()*. This causes sevenstaxTCP to listen on a specified port for incoming connections.
- After being connected, which will be indicated by the received notify code *NC_TCP_CONNECTED,* call *stxTCP_WriteBulk()* to send data of nearly any size to the remote TCP.
- When the entire data block was successfully transmitted, more data may be sent or the connection may be closed by calling *stxTCP_Disconnect()* for a graceful disconnect.

8.1.7 Single and Multiple Connections

In cases, where memory is a very limited resource, sevenstaxTCP can work in “single connection mode”. This is useful for very small embedded devices with the need of only one TCP connection at a time. Set *TCP_MULTI* to 0 or 1 to enable this mode.

Typically more than one simultaneous connection must exist (e.g. HTTP browser requests). sevenstaxTCP can be configured to work with up to 65535 connections at a time. Multi connection support can be switched on at compile time by setting *TCP_MULTI* to a value of at least 2.

With *TCP_MULTI* one can specify the maximum number of connections which sevenstaxTCP can handle. This should be well adjusted to save memory because for every connection about 100 bytes RAM are reserved.

TCP_MULTI_TICKS is the number of active connections which are handled by a single call to *stxTCP_Tick()*. This value also depends a little how frequently *stxTCP_Tick()* is called. For example: If you configure sevenstaxTCP to work with 100 possible connections, TCP_MULTI_TICKS can be set to 10.

Both defines can be found in '*tcpdefs.h*'.

Most of sevenstaxTCP's interfaces have an additional parameter called 'Connection-ID'. Connection IDs are unique values which help to distinguish between connections. They are unique during the lifetime of a connection. Connection IDs are automatically recycled (newly assigned) when a connection ends.

Please Note: *TCP_MULTI_TICKS* does not affect incoming TCP data. Received packets are handled immediately, whenever the underlying protocol layer (probably IP) calls *stxTCP_ProcessPacket()*.

8.2 Protocol API

8.2.1 stxTCP_Init (handler)

Description

Before sevenstaxTCP can be used, this function must be called. It initializes all internal states of sevenstaxTCP and also registers the application-defined handler procedure.

Parameter

PROTOCOL_NOTIFY_HANDLER *handler*
Application handler function to receive notify codes from sevenstaxTCP.

Return Value

-

Comment

Please Note: It is advisable to initialize the link layer and set up the receive and transmit packet descriptors *before* calling *stxTCP_Init()*.

8.2.2 stxTCP_Tick ( )

Description

This function keeps sevenstaxTCP alive. It must be called as often as possible in order to provide sevenstaxTCP with processor time.

It is designed to be very fast thus applications are not slowed down. The most time-consuming action might be to build an entire TCP packet (*TCP_MULTI_TICKS* per call) and deliver it to the link layer. Therefore, the maximum time *stxTCP_Tick()* will use depends on the size of the biggest packet (defined by *RTM_MAXDATA*).

Parameter

-

Return Value

-

Comment

-

8.2.3 stxTCP_Connect (hSubConn, ip_address, port)

Description

Tries to establish a connection to another TCP.

Parameter

CONN_HNDL_TYPE *hSubConn* (only multi device version)
Connection handle of underlying network layer. If NULL_stx no restriction is made concerning the local IP subnet.

CONST_stx IPV4 FPTR_stx *ip_address*
A pointer to an IPV4 structure which must be initialized to the IP address of the remote peer.

UINT16_stx *port*
The port number of the remote TCP on which the connection should be established. Ports are something like channels dedicated to particular services. E.g. HTTP resides on port 80 (decimal).

Return Value

UINT16_stx
The sevenstaxTCP internal connection ID, or an error value (*INVALID_CONNECTION_ID*).

Subsequent calls into sevenstaxTCP must use this value to identify the connection. Also events, which the application receives by its handler procedure, carry connection IDs.

Comment

Upon a call to *stxTCP_Connect()*, successful or not, the application-defined handler procedure is called. *stxTCP_Connect()* is usually called when sevenstaxTCP acts as a client TCP. Server TCPs should call *stxTCP_Listen()* instead.

8.2.4 stxTCP_Listen (hSubConn, port)

Description

Puts sevenstaxTCP into the listen state, which means that it waits for a connection attempt on the specified port. If another TCP sends a SYN, sevenstaxTCP handles it properly and enters the connected state.

Parameter

CONN_HNDL_TYPE *hSubConn* (only multi device version)
Connection handle of underlying network layer. If NULL_stx no restriction is made concerning the local IP subnet.

UINT16_stx *port*
The port number of the local TCP, on which the connection should be established. Ports are something like channels dedicated to particular services. E.g. HTTP resides on port 80 (decimal).

Return Value

UINT16_stx
The sevenstaxTCP internal connection ID, or an error value (*INVALID_CONNECTION_ID*). Subsequent calls into sevenstaxTCP must use this value to identify the connection. Also events, which the application receives by its handler procedure, carry connection IDs.

Comment

stxTCP_Listen() only succeeds, if a free TCP instance is found.

8.2.5 stxTCP_WriteBulk (uConnId, data, len, SendDynamic)

Description

When a connection is established, *stxTCP_WriteBulk()* can be used to send data to the remote station. This function returns immediately and sends data asynchronously the next time *stxTCP_Tick()* is called.

Parameter

UINT16_stx *uConnId*
connection ID where to send. Can be obtained either by a call to *stxTCP_Connect()* or *stxTCP_Listen()*.

UINT8_stx FPTR_stx *data*
Pointer to the data to send, points to the first byte of the data that should be sent.

UINT32_stx len
Size of the data in bytes. If *bSendDynamic* == *TRUE_stx* is used, this length should be at least the number of bytes assumed to be sent (see comment below).

BOOL_stx *bSendDynamic*
Get data to send via notify handler. Set it to *FALSE_stx* per default. *TRUE_stx* might be used by an application, which dynamically generates send data instead of simply giving a const data pointer. sevenstaxTCP will repeatedly call the application notify handler with notify code *NC_TCP_GET_TXDATA* to request data to send. See section 8.3.6 for details.

Return Value

BOOL_stx
TRUE_stx, if *stxTCP_WriteBulk()* has accepted a request to send. Otherwise

FALSE_stx. Succeeds only if a connection is established and no transmission is currently in progress.

Comment

Because sevenstaxTCP is designed to run on small embedded systems, it does not copy the data but operates on the source to avoid wasting memory. This function is able to send huge amounts of data to the remote station. Data is fragmented into *RTM_MAXDATA* sized pieces and internally sent out using multiple calls. *NC_TCP_TXREADY* is issued only after the last fragment was successfully transmitted (and acknowledged).

Please Note: The data (and possibly a reference) must remain constant until *stxTCP_WriteBulk()* has finished (which is signaled by an *NC_TCP_TXREADY* event, see below).

8.2.6 stxTCP_Disconnect (uConnId, bForced)

Description

This function can be called by the application to end a TCP connection. sevenstaxTCP does all necessary actions to close a TCP connection gracefully.

The application receives the result as a call to its handler procedure (*NC_TCP_FIN*).

Parameter

UINT16_stx *uConnId*
connection ID of TCP connection to be closed. connection ID was returned by *stxTCP_Connect()* or *stxTCP_Listen()*.

BOOL_stx *bForced*
Default *FALSE_stx* should be used to support normal TCP disconnection states. *TRUE_stx* forces an immediate RST. Might be useful in case of any connection problems.

Return Value

BOOL_stx
TRUE_stx if a close can be performed, *FALSE_stx* otherwise

Comment

Usually *stxTCP_Disconnect()* only fails when the connection is already in its closed state.

8.2.7 stxTCP_GetState (uConnId)

Description

This function can be called by the application to get the state of a TCP connection.

Parameter

UINT16_stx *uConnId*
connection ID of TCP connection.
connection ID was returned by *stxTCP_Connect()* or *stxTCP_Listen()*.

Return Value

TCP_STATUS
Status of TCP connection

Comment

-

8.3 Notify Codes

With one of the NC_TCP_XXX codes, sevenstaxTCP will call the handler procedure to inform about occurred events.

Some notify codes transfer a pointer to a TCP information structure.

This structure is mainly used to pass all important properties regarding to a TCP connection. E.g. when sevenstaxTCP has posted an *NC_TCP_CONNECTED* event to the handler routine of the application, some members (remote_ip_address, remote_port and local_port) contain values which describe the connection.

```
typedef struct _tag_TCP_INFORMATION
{
    UINT8_stx   FPTR_stx data;      /* Points to received data */
    IPV4        remote_ip_address;  /* IP address of peer */
    IPV4        dest_ip_address;    /* our IP / or broadcast / multicast  */
    UINT16_stx  datasize;           /* Length of received data */
    UINT16_stx  remote_port;        /* Port number of peer */
    UINT16_stx  local_port;         /* Our port number */
} TCP_INFORMATION;
```

UINT8_stx FPTR_stx data

Points to received data. The application must only use this pointer and the data referenced to, as long as the called function (notify handler) is active.

IPV4 remote_ip_address

Remote IP address (sender), might be Unicast, Multicast or Broadcast.

IPV4 dest_ip_address

Destination IP address (receiver), might be Unicast, Multicast or Broadcast.

UINT16_stx datasize;

Amount of data in bytes, *data* points to.

UINT16_stx remote_port;

IP-Port number of the remote (port of sender).

UINT16_stx local_port

IP-Port number of the destination (port of receiver)

If data arrives on a specific port, sevenstaxTCP issues a *NC_TCP_RECV* event with an additional pointer to a *TCP_INFORMATION* structure where (beside the other members) data and datasize describe a newly received packet.

Below is a description of the notify codes.

8.3.1 NC_TCP_CONNECTED (TCP-Info, ConnID)

After calling *stxTCP_Connect()*, *NC_TCP_CONNECTED* is issued in case of success. The application can now send and receive data to/from sevenstaxTCP. Both parameters contain port numbers copied directly from the received packet.

Parameter 1 points to a *TCP_INFORMATION* structure, which describes the newly created connection. The members 'remote_ip_address', 'remote_port' and 'local_port' are valid.

Parameter 2 contains a Connection-ID, which has been returned by *stxTCP_Connect()*.

ReturnValue not used.

8.3.2 NC_TCP_CONNTIMEOUT (TCP-Info, ConnID)

This event occurs if *stxTCP_Connect()* failed (i.e. timed out). sevenstaxTCP does a few retries to establish a connection and then gives up.

Parameter 1 points to a *TCP_INFORMATION* structure, which describes connection. The members '*remote_ip_address*', '*remote_port*' and '*local_port*' are valid.

Parameter 2 is a Connection-ID, which identifies the connection that failed.

ReturnValue not used.

8.3.3 NC_TCP_FIN (AbortReason, ConnID)

This event is posted when the TCP connection is terminated. This may happen when a web server has finished sending data, when an error occurs, or after the application has called *stxTCP_Disconnect()*.

Parameter 1 is a *ABORT_REASON* (see section 8.1.5, Terminating a TCP Connection)

Parameter 2 is a Connection-ID, which identifies the connection that has been closed.

ReturnValue not used.

8.3.4 NC_TCP_RECV (TCP-Info, ConnID)

When sevenstaxTCP has received a data packet, *TCP_RECV* is issued to tell the application, that data has arrived.

Parameter 1 points to a *TCP_INFORMATION* structure which describes the incoming data packet. All members are valid.

Parameter 2 is a Connection-ID, which identifies the connection.

ReturnValue not used.

8.3.5 NC_TCP_TXREADY (TCP-Info, ConnID)

This event is posted to signal the application, that sevenstaxTCP is ready to accept data for next transmission. It occurs after sevenstaxTCP has received an ACK as an answer to the transmitted data.

After getting *NC_TCP_TXREADY*, the buffer allocated for *stxTCP_WriteBulk()* can be reused/freed. *NC_TCP_TXREADY* is issued after sevenstaxTCP has transmitted the entire range of data which was specified through a call to *stxTCP_WriteBulk()*.

Parameter 1 points to a *TCP_INFORMATION* structure, which describes the TCP connection. The members '*remote_ip_address*', '*remote_port*' and '*local_port*' are valid.

Parameter 2 is a Connection-ID, which identifies the connection.

ReturnValue not used.

8.3.6 NC_TCP_GET_TXDATA (TCP_TxData, ConnID)

If *stxTCP_WriteBulk()* was called with parameter *bSendDynamic* == *TRUE_stx* (see section 8.2.5 for details), sevenstaxTCP will call the notify handler with *NC_TCP_GET_TXDATA,* to get the next data block to send. The application now is responsible to copy the requested data into the given destination address. This will repeatedly be done until the application notify handler returns 0 for this call.

Although this feature might be used rarely, this gives the advantage, that the application dynamically might send a very large amount of data, without the need to keep the complete databuffer secure until *NC_TCP_TXREADY* is received. On the other hand, if there is any TCP retransmission necessary, the application must be able to generate this dynamic data again.

Parameter 1 is a pointer to a *TCP_TXDATA* structure, which contains sevenstaxTCP instructions for the application: *pSourceData* is the application source data pointer, which sevenstaxTCP wants to send next. Normally this pointer will increase with every call. But it will be set back, if e.g. a TCP retransmission is necessary. Please assure, that your application is able to set this source pointer back to old data! *pTargetBuffer* is the target address, to which the application should copy the data. *uCount* is number of bytes, which sevenstaxTCP expects to receive.

Parameter 2 is a Connection-ID, which identifies the connection.

ReturnValue returns the number of bytes, the application has copied from *pSourceData* into pTargetBuffer. If the last data has been sent, the application should return 0 here.

8.4 Adjustable Parameters

There are some parameters defined in '*tcpdefs.h*' which can be customized. These parameters should properly be adjusted depending on the lower network layer and communication device. The following list describes them in detail.

TCP_MULTI **default: 5** **valid: 0..n**

If multiple TCP connections are required, this specifies the maximum number of simultaneous connections. For a single connection TCP, set this parameter to 0.

Please keep in mind, that RAM requirements increase by about 100 Bytes per defined TCP_MULTI instance.

TCP_MULTI_TICKS **default: 1** **valid: 1..n**

If multiple connections are needed, this specifies the number of connections which are handled by a single call to *stxTCP_Tick()*. This value is ignored, if TCP_MULTI is set to 0.

RTM_TIMEOUT **default: 5000**

Retransmission timeout. This is the time in milliseconds within a transmitted packet must be acknowledged. If sevenstaxTCP does not receive an ACK, the packet will be resent.

RTM_MAXRETRIES **default: 12**

This is the number of attempts sevenstaxTCP will make to transmit a single packet to the other end.

TCP_CONNECTION_RETRIES **default: 10**

This is the number of attempts sevenstaxTCP will make to establish a connection.

TCP_SYN_REPEAT **default: 3000**

Delay in ms between SYN TX retries while establishing connection

TCP_FINACK_TIMEOUT **default: 20000**

Timeout in ms to wait for ACK of remote to our FINACK

TCP_FINACK_REPEAT **default: 3000**

Delay in ms between FINACK TX retries while closing connection

TCP_KEEP_ALIVE_HANDLING **default: 0 (= off)**
valid: 1 (= sending keep-alive packets)
2 (= automatic close)

This parameter controls generation of additional optional keep-alive TCP packets and handling of long time connections without traffic. Some application or connection types might need this feature to validate an open TCP connection.

If set to 0, no keep-alive handling nor connection surveillance is executed. This is the default behaviour.

If set to 1, keep-alive packets will automatically be sent to the remote client in *TCP_KEEP_ALIVE_INTERVAL* seconds.

If set to 2, no keep-alive packets will be sent. But TCP connections will automatically be closed after *TCP_KEEP_ALIVE_INTERVAL* seconds without traffic.

TCP_KEEP_ALIVE_INTERVAL **default: 7200**

If *TCP_KEEP_ALIVE_HANDLING* is set to 1, this is the time interval in seconds to repeat sending keep-alive TCP packets.

If *TCP_KEEP_ALIVE_HANDLING* is set to 2, this is the time interval in seconds after which an open TCP connection will be closed, because of no traffic.

TCP_CHECK_CRC **default: 1**

Enables checking the checksum of received packets.

RTM_MAXDATA **default: Max. size of TCP send payload**

The maximum size in bytes of a single TCP packet to be sent.

Of course *stxTCP_WriteBulk()* handles larger amount of data by fragmenting it into several packets of *RTM_MAXDATA* byte size.

With the default value exactly one packet fits into the send buffer.

TCP_WINDOW_SIZE **default: Max. size of TCP receive payload - overhead to save a rcv. TCP packet**

This stands for the maximum number of bytes a TCP can receive. This value will be directly written to the TCP header. Normally, a TCP sets this value dynamically according to its free buffer space.

With the default value exactly two complete packets fit into the receive buffer.

9 UDP Module

9.1 Overview

UDP is part of sevenstaxTCP/IP standard internet library. It can be enabled by setting *UDP_SUPPORTED* to 1 in *features.h* (see section 2, Supported Protocols).

9.2 Protocol API

9.2.1 stxUDP_Init (handler)

Description

Sets up some internal variables, initialises internal UDP instances and registers application callback function. Must be called before using any other UDP function.

Parameter

PROTOCOL_NOTIFY_HANDLER *handler*
Application handler function to receive notify codes from the UDP module. The handler function will be called, when UDP data is received for a port that was not registered via stxUDP_AddConn() (see below).

Return Value

-

Comment

-

9.2.2 stxUDP_Tick ( )

Description

This function keeps sevenstaxUDP alive. It must be called as often as possible in order to provide sevenstaxUDP with processor time.

In this version the function does nothing.

Parameter

-

Return Value

-

Comment

-

9.2.3 stxUDP_AddConn (hSubConn, local_port, fProcessFunction)

Description

Calling this function will enable one of the possible *UDP_MULTI* UDP instances to be used for sending and/or listening UDP packets.

If *fProcessFunction* is NULL_stx, any incoming UDP data will be transferred to the application via the notify code *NC_UDP_RECV*.

If *fProcessFunction* is != NULL_stx, incoming UDP data will directly be transferred to the given process function. The application will not be notified.

Parameter

CONN_HNDL_TYPE *hSubConn* (only multi device version)
Connection handle of underlying network layer. If NULL_stx no restriction is made concerning the local IP subnet.

UINT16_stx *local_port*
Local UDP port that is to be activated (active sending and passive listening).

UDP_PROCESS_FCT *fProcessFunction*
The application or stack internal function, which will be called directly with a UDP information structure as parameter to process the UDP data. If being set to NULL_stx, the standard UDP notify handler will be called with notify code NC_UDP_RECV and a UDP information structure.
The UDP information structure is described below in the chapter “Notify Codes”

Return Value

UINT16_stx
UDP connection ID, to be used for further UDP functions. If the same UDP port (and in multi device version same sub connection handle) has already been 'registered' by previous calls, this function will return the same connection ID as before. Returns *UDP_INVALID_ID*, if there are no more free UDP instances (defined by *UDP_MULTI*).

Comment

Please note that it is possible to independently use TCP ports and UDP ports at the same time, on both either server (listening) or client (acting) mode.

9.2.4 stxUDP_DelConn (uConnId)

Description

Calling this function disables listening or sending on the given UDP connection ID. The corresponding UDP port will be closed.

Parameter

UINT16_stx *uConnId*
UDP connection ID, returned by *stxUDP_AddConn()*.

Return Value

-

Comment

-

9.2.5 stxUDP_Write (uConnId, psRemoteIp, uRemotePort, data, datalen)

Description

Immediately sends a UDP datagram to another host.

Parameter

UINT16_stx *uConnId*
Valid UDP connection ID, returned by *stxUDP_AddConn()*.

IPV4 FPTR_stx *psRemoteIp*
The IP-Address of the target.

UINT16_stx *uRemotePort*
The UDP port of the target.

UINT8_stx FPTR_stx *data*
Pointer to the data to be inserted as UDP payload.

UINT16_stx *datalen*
Amount of payload bytes to be send.

Return Value

BOOL_stx
TRUE_stx if data has been sent, *FALSE_stx* if any error.

Comment

stxUDP_AddConn() must have been called before using this function.

9.3 Notify Codes

One notify code transfers a pointer to a UDP information structure.

This structure is mainly used to pass all important properties regarding to a UDP connection. When sevenstaxUDP has posted an *NC_UDP_RECV* event to the handler routine of the application, some members contain values which describe the connection.

```
typedef struct _tag_UDP_INFORMATION
{
    IPV4        remote_ip_address;  /* IP address of peer */
    IPV4        dest_ip_address;    /* our IP / or broadcast / multicast  */
    UINT16_stx  remote_port;        /* Port number of peer */
    UINT16_stx  dest_port;          /* Our port number */
    UINT16_stx  datasize;           /* Length of received data */
    UINT8_stx   FPTR_stx data;      /* Points to received data */
} UDP_INFORMATION;
```

UINT8_stx FPTR_stx data

Points to received data. The application must only use this pointer and the data referenced to as long as the called function (notify handler) is active.

IPV4 remote_ip_address

Remote IP address (sender), might be Unicast, Multicast or Broadcast.

IPV4 dest_ip_address

Destination IP address (receiver), might be Unicast, Multicast or Broadcast.

UINT16_stx datasize;

Amount of data in bytes, *data* points to.

UINT16_stx remote_port;

IP-Port number of remote (port of sender).

UINT16_stx local_port

IP-Port number of destination (port of receiver)

Description of the notify codes:

9.3.1 NC_UDP_RECV (UDP-Info, UDPConnID)

This event is generated as a result of a successfully received UDP packet from the remote machine. It is called only if the target port could be found in the active connections (see also: *stxIP_AddConn()*) and if the appropriate process function is NULL_stx.

Parameter 1 points to a UDP_INFORMATION structure, which describes the incoming data packet. All members are valid.

Parameter 2 indicates the ID of the UDP instance, on which this data was received.

ReturnValue not used.

9.4 Adjustable Parameters

UDP parameters are defined in '*udpdefs.h*'. These parameters should properly be adjusted depending on the application requirements. The following list describes them in detail.

UDP_MULTI default: 5 valid: 1..n

If multiple UDP transmissions are required, this specifies the maximum number of simultaneous UDP connections. Only 4 Bytes RAM are allocated per *UDP_MULTI*.

Please keep in mind, that DNS, BOOTP, DHCP or mDNS need their own free UDP instances, if they are enabled.

UDP_CHECK_CRC default: 1

Enables checking the checksum of received packets.

10 DNS Tiny Module

10.1 Overview

A simple DNS client is part of sevenstaxTCP/IP standard Internet library. The application can send a DNS query to get the IP address of server with a known name. DNS can be enabled by setting *DNS_SUPPORTED* to 1 in *features.h* (see section 2, Supported Protocols).

The DNS protocol works ahead UDP, so *UDP_SUPPORTED* must be set to 1 too (see section 2, Supported Protocols).

For humans it is more convenient to work with readable names instead of IP addresses. For example the URL 'www.yahoo.com' to access a website instead of using the IP address 64.58.79.230 is much easier to memorize. That is the reason why DNS was created. A DNS or domain name service, converts names to their corresponding IP addresses.

The modern DNS is organised a bit like a tree. There is a root called '.' with certain main branches called 'top level domains' (TLD). At the root are a few servers (e.g. a.root-servers.net, b.root-servers.net ...) which know all addresses of all the servers which handle individual TLDs (Top Level Domains) such as .com, .edu, .mil or .net.

Communication with the DNS is done by UDP on port 53. DNS client software (so called resolvers) must send special packets (see RFC 1035) to the DNS server. The DNS server, on his part, sends back a packet containing some records which contain e.g. the IP address.

To use sevenstaxTCP/IP's integrated DNS resolver within your application, the following steps are required:

- In *features.h* set *DNS_SUPPORTED* to 1.
- Because the DNS protocol needs UDP as the transport layer, set *UDP_SUPPORTED* to 1 in *features.h* too. Also assure at least 1 free UDP instance with *UDP_MULTI* in 'udp_defs.h'.
- The address of the DNS-server can be retrieved via the Internet Service Provider (if PPP is used), via the DHCP Server (if Ethernet is used), or manually. The primary DNS server should be used preferentially. The secondary DNS server should be used only if the primary DNS server is down.
- Call *stxDNS_Init()* once to set-up sevenstaxDNS internal state.
- After initialization of DNS call *stxDNS_QueryStart()* supplied with the IP address of the DNS server and the name of the server that will be searched. The DNS local port is set automatically by the sevenstax software.
- In your main loop or operating system task call *stxDNS_Tick()* repeatedly in order to keep alive and finish DNS queries.
- On completion, sevenstaxTCP/IP will notify the application either with *NC_DNS_IP* or *NC_DNS_TIMEOUT*.
- When receiving the event *NC_DNS_IP*, which means that the DNS server has answered, the delivered IP address can be contacted.

10.2 Protocol API

10.2.1 stxDNS_Init (handler)

Description

Sets up some internal variables and registers the application callback function. Must be called before using any other DNS function.

Parameter

PROTOCOL_NOTIFY_HANDLER handler
Application handler function to receive notify codes from sevenstaxTCP/IP's DNS module. The handler function will be called, when DNS data is received or a timeout occurs

Return Value

-

Comment

-

10.2.2 stxDNS_Tick ( )

Description

This function keeps the DNS query alive, assuring multiple retries until a timeout occurs. It should repeatedly be called by the application, like all the other sevenstax “Tick” functions too, independently of a pending DNS query or not.

Parameter

-

Return Value

-

Comment

In case of a timeout, this function will call the application notify handler with *NC_DNS_TIMEOUT*. If the DNS server responds, this will be synchronously be signaled by *NC_DNS_IP* by the responsible process-packet-functions.

10.2.3 stxDNS_QueryStart (hConnNetwork, stx dns_ip, www_name)

Description

Instructs sevenstaxTCP/IP to transmit DNS queries on UDP port 53 to a known DNS server. Multiple DNS queries will be sent by *stxDNS_Tick()*. The result is posted as a notification code to the handler procedure, which can either be *NC_DNS_IP* (success) or *NC_DNS_TIMEOUT* (failure).

Parameter

CONN_HNDL_TYPE *hConnNetwork* (only multi device version)
connection handle of underlying network layer over which the DNS query ought to be started.

CONST_stx IPV4 FPTR_stx *dns_ip*
IP address of the DNS server to be used.

STRING_stx *www_name*
A pointer to the web-name, e.g. “www.yahoo.com”.

Return Value

BOOL_stx
TRUE_stx, if the DNS query packet was successfully delivered over UDP.
FALSE_stx, if a DNS query is already in progress, this means that neither the DNS server answered nor a timeout occurred.

Comment

This function internally uses the UDP API of sevenstaxTCP/IP. Although there are no additional function calls necessary, please assure, that at least one free UDP instance must be available.

10.2.4 stxDNS_QueryStop (hConnNetwork)

Description

Stops a pending DNS query and releases UDP port.

Parameter

CONN_HNDL_TYPE *hConnNetwork* (only multi device version)
connection handle of underlying network layer over which the DNS query ought to be stopped.

Return Value

-

Comment

There is normally no need to call this function, because notify codes will be delivered, if a timeout occurs. But if needed, DNS immediately will stop any query by calling this function, no notify code will be delivered.

10.3 Notify Codes

10.3.1 NC_DNS_IP (IP addr, -)

After calling *stxDNS_QueryStart()*, this event occurs when the DNS client has received a valid answer and a record is found, which contains the requested IP address.

Parameter 1 contains the IP address, passed in network byte order.

Parameter 2 is not used and always UINT32_stx_MAX.

ReturnValue not used.

10.3.2 NC_DNS_TIMEOUT (- , -)

This event is posted as a negative result when *stxDNS_QueryStart()* was called and the DNS resolver timed out.

There are various reasons for such a failure. The most likely reasons are, that the DNS server is not able to find the requested host name (e.g. because it does not exist anymore), or that the DNS server itself is temporarily not available.

Parameter 1 is not used and always UINT32_stx_MAX.

Parameter 2 is not used and always UINT32_stx_MAX.

ReturnValue not used.

10.4 Adjustable Parameters

Currently there is no DNS parameter available for this tiny DNS client module.

11 ICMP Module

11.1 Overview

A simple ICMP responder is part of sevenstaxTCP/IP standard internet library. sevenstaxICMP automatically responds to an incoming “ping” with an ICMP packet.

ICMP can be enabled by setting *ICMP_SUPPORTED* to 1 in *features.h* (see section 2, Supported Protocols).

If an incoming packet is identified as an ICMP echo request, sevenstaxICMP does the following steps:

- The packet is copied to the TX buffer and the type is changed to an ICMP echo reply.
- A new checksum is calculated.
- Source and Destination addresses are swapped.
- The packet is sent off to the network.

11.2 Protocol API

Currently there is no public ICMP API function available, nor a function to send an ICMP message by the application.

11.3 Notify Codes

Currently there is no notify code available.

11.4 Adjustable Parameters

Currently there is no ICMP parameter available.

12 NTP Module

12.1 Overview

A simple NTP client is part of sevenstaxTCP/IP standard Internet library. NTP can be used to retrieve the current time.

NTP can be enabled by setting *NTP_SUPPORTED* to 1 in *features.h* (see section 2, Supported Protocols).

The NTP protocol works ahead UDP, so *UDP_SUPPORTED* must also be set to 1 (see section 2, Supported Protocols).

The Network Time Protocol (NTP) is used to synchronize the time of a computer client or server to another server or reference time source, such as a radio or satellite receiver or modem. It provides accuracies typically within a millisecond on LANs and up to a few tens of milliseconds on WANs relative to Coordinated Universal Time (UTC) via a Global Positioning Service (GPS) receiver, for example. Typical NTP configurations utilize multiple redundant servers and diverse network paths in order to achieve high accuracy and reliability.

NTP uses UDP on port 123 for both, transmission and reception.

To access NTP servers with sevenstaxNTP you should proceed in the following way:

1. Set *NTP_SUPPORTED* to 1 in *features.h*
2. Call *stxNTP_Init()* once to set-up sevenstaxNTP's internal state.
3. Call *stxNTP_Query()* once to start querying the NTP server.
4. In your main loop or operating system task call *stxNTP_Tick()* repeatedly in order to keep alive and finish NTP queries.
5. After a while, the handler procedure receives either an *NC_NTP_SUCCESS* or *NC_NTP_FAILURE* event.
6. On *NC_NTP_SUCCESS* call *stxNTP_Transform()* with the 32-bits value in order to convert NTP output into human suitable time values.
7. On *NC_NTP_FAILURE* the query can, for example, be repeated with another time server's IP address.

12.2 Protocol API

12.2.1 stxNTP_Init (handler)

Description

Sets up some internal variables and registers the application callback function. Must be called once before using any other NTP function.

Parameter

PROTOCOL_NOTIFY_HANDLER *handler*
Application handler function to receive notify codes from sevenstaxNTP module. The handler function will be called, when NTP data is received or a timeout occurs.

Return Value

void

Comment

-

12.2.2 stxNTP_Tick ( )

Description

This function keeps NTP alive, assuring multiple retries until a timeout occurs. It should repeatedly be called by the application, like all the other sevenstax “Tick” functions too, independently of a pending NTP query or not.

Parameter

-

Return Value

-

Comment

In case of a timeout, this function will call the application notify handler with *NC_NTP_FAILURE*. If the NTP server responds, this will be synchronously be signaled by *NC_NTP_SUCCESS* by the responsible process-packet-functions.

12.2.3 stxNTP_Query (hConnNetwork, ntp_server)

Description

NTP queries will be started by calling this function. Sets some internal variables and returns immediately. The actual query will be done the next time *stxTCP_Tick()* is called. After a call to *stxNTP_Query()*, the application defined handler procedure will either receive an *NC_NTP_SUCCESS* (got the time) or *NC_NTP_FAILURE* (no response from the time server).

Parameter

CONN_HNDL_TYPE *hConnNetwork* (only multi device version)
connection handle of underlying network layer over which the NTP query ought to be started.

CONST_stx IPV4 FPTR_stx *ntp_server*
A pointer to an IPV4 structure representing the IP address of the time server.

Return Value

BOOL_stx
TRUE_stx, if the query was started
FALSE_stx, if the query could not be started, because of no free UDP instance

Comment

Please note: Calls made to *stxNTP_Query()* while being in progress will re-trigger the query process.

12.2.4 stxNTP_Transform (ntp_seconds, seconds, minutes, hours, year, month, day, day_of_week *)*

Description

Takes the NTP seconds (ntp_seconds) delivered by the time server and writes the results into the locations for application time variables.

Parameter

UINT32_stx *ntp_seconds*
NTP seconds delivered by the time server

UINT8_stx FPTR_stx *seconds,* UINT8_stx FPTR_stx *minutes,*
UINT8_stx FPTR_stx *hours,* UINT16_stx FPTR_stx *year,*
UINT8_stx FPTR_stx *month,* UINT8_stx FPTR_stx *day,*
UINT8_stx FPTR_stx *day_of_week*
Addresses of application variables to receive the delivered time values.

Return Value

-

Comment

This function is for convenient conversion of the result of an NTP query (which is a 32-bit value of seconds since 1.1.1900) into time and date values.

The time returned by the servers is UTC (Coordinated Universal Time) which means, it has to be corrected if the local time zone differs from UTC. This correction must be done before calling *stxNTP_Transform()*. The difference between the local time and UTC has to be added to the seconds returned by the server. After calling *NTP_Transform()* the right values can be read from the parameters as described above.

12.3 Notify Codes

12.3.1 NC_NTP_SUCCESS *(NTP-sec, -)*

This event indicates a successful reception of a time server response.

Parameter 1 contains the time in seconds since 01.01.1900. This value might be converted by the application with *stxNTP_Transform()*.

Parameter 2 is not used and always UINT32_stx_MAX.

ReturnValue not used.

12.3.1 NC_NTP_FAILURE *(- , -)*

This is the opposite to NC_NTP_SUCCESS. When an NTP query was made, NC_NTP_FAILURE is posted to the handler procedure, if the time server does not respond properly.

Parameter 1 is not used and always UINT32_stx_MAX.

Parameter 2 is not used and always UINT32_stx_MAX.

ReturnValue not used.

12.4 Adjustable Parameters

Currently there is no NTP parameter available.

13 Debugging Support

At source level, sevenstaxTCP/IP contains a simple debugging support through macros mapped to printf() (DebugOut-Strings).

The main switch to enable DebugOuts for all modules is DEBUG_STX = 1 in *features.h*.

```
/* application */
#define DEBUG_STX               1       /* Main switch: sevenstax debugging enabled */
```

Each module has a separate switch to enable debug information at compilation time. This enables a fine control about the debug information you will get. You can control it through the settings in *features.h*:

```
...
/* protocols */
#define IP_DEBUG                1       /* ipv4 module */
#define TCP_DEBUG               1       /* tcp module */
#define TCPPACKET_DEBUG         1       /* tcp module, packet details */
#define TCPPACKET_DUMP          1       /* tcp module, packet content */
#define UDP_DEBUG               1       /* udp module */
#define DNS_DEBUG               1       /* dns & dns_srv module */
#define DNS_DETAILS             1       /* dns & dns_srv & mdns packet details */
#define ICMP_DEBUG              1       /* icmp module */
#define HTTPSERV_DEBUG          1       /* httpserv & html_page modules */
#define NTP_DEBUG               1       /* ntp module */
...
```

Each module has a separate macro to map DebugOuts to a common printf() function:
(e.g. tcpdefs.h):

```
...
/* control debug out */
#if (TCP_DEBUG==1 && DEBUG_STX==1)
    #include "debugstx.h"

    #define TCPPRINT(a)                     DBGPRINTLN("TCP",a)
    #define TCPPRINT1(a,b)                  DBGPRINTLN1("TCP",a,b)
    #define TCPPRINT2(a,b,c)                DBGPRINTLN2("TCP",a,b,c)
...
```

All DebugOuts are mapped to the central DBGOUTFCT macro in 'debugstx.h'. This is the point, where the developer might map to an alternative to printf(), e.g. to another serial out function or save it in memory.

Details of DebugOut might be controlled via define switches in *features.h*:

```
/* configure format  */
#define DBGFMT_TIMESTAMP        1       /* enables preceding timestamp  */
#define DBGFMT_FILELINE         1       /* enables preceding filename/line number */
#define DBGFMT_MODULE           1       /* enables preceding module name like  */
```

14 Restrictions

sevenstaxTCP/IP is focused on systems with very small resources, to give them relevant TCP/IP functionality. Therefore – in comparison to a full featured stack implementation of for example a PC software - it has some restrictions in usage and capabilities that the user should know:

#	*Restriction*	*Details*
1	Not aware of interrupts	Per default, sevenstaxTCP/IP functions are called inside a “tick” function, which is normally located inside an endless while() loop or OS task. If parts of sevenstax embedded protocols are called from within an ISR, synchronization must be done by the user.
2	Not full RFC compliant	To give maximum capabilities within small system resources, sevenstax embedded protocols fulfill only basic and most relevant parts of the RFC specifications.
3	No buffering of any kind	Because sevenstaxTCP/IP has no data buffers, when sending data over a TCP connection using stxTCP_WriteBulk(), the content must exist and must not be changed, until the application receives a feedback (NC_TCP_TXREADY).
4	IP fragmentation	sevenstaxTCP/IP does not support IP fragmentation. This is no disadvantage for TCP/IP, but for UDP packets: Only the first part of fragmented UDP packets are received and used. sevenstaxTCP/IP also never generates fragmented IP packets by itself.
5	Replies from name servers (DNS)	Because DNS packets are sent over UDP and sevenstaxTCP/IP throws away UDP fragments, DNS only works if either the reply (from the name server) is not fragmented or the wanted IP address is in the first fragment. For very large sites, such as www.google.com, name server replies does not fit into a single UDP packet. But in the majority of cases, an IP address is included and found in the first fragment.
6	IP address from name servers (DNS)	sevenstaxTCP/IP only uses the first of (potential multiple) IP address found in the reply of a name server.
7	ICMP	The only part of ICMP, which is currently implemented into sevenstaxTCP/IP, are ICMP echo replies (PING).
8	Use of Ethernet and PPP	The single device version of sevenstaxTCP/IP does not support use of PPP mode and Ethernet mode at the same time. Please use 'sevenstaxTCP/IP MultiDevice', if you need several physical interfaces at the same time (e.g. router functionality).

sevenstaxTCP/IP is object of permanent improvement. Please contact info@sevenstax.de to get more information about new TCP/IP features that will be available in next release version.

15 Change History

Ver.	*Date*	*by*	*Change description*
2.88	21. Aug. 2002	pe	Base version
2.91	12. Dec. 2002	pe	Improved API documentation
3.01	16. Jan. 2003	pe	Additional features added
3.02	13. Feb. 2003	krz	Completely reviewed
3.03	21. Feb. 2003	krz	Some small corrections
3.11	26. Feb. 2003	pe	Restrictions reviewed
3.12	22. July 2003	ck	New graphics
3.13	24. May 2004	krz	Completely reviewed
3.13a	27. May 2004	krz	ARP and NotifyCodes updated
3.14	01. June 2004	krf	Released
3.15	29. June 2004	krz	Format review
3.16	30. July 2004	krz	Additional TCP Information in NotifyCodes, ARP restrictions removed, types extended by '_stx' prefix, corrections
3.17	13. Aug. 2004	ck	Proof-Read
3.18	20. Aug. 2004	krz	Added header dependencies
3.30	04. May 2006	krz	Reviewed for sevenstaxTCP V3.3
4.20	12. Mar 2007	krz	Completely reviewed for sevenstaxTCP V4.2 - several new and changed API functions - new graphics
4.21	20. Oct 2007	krz	UDP API changes
5.00	03. Mar 2008	ssu	Completely revised for sevenstaxTCP/IP & sevenstaxRouting V5.00
5.01	10. Mar 2008	jma	Small corrections
5.02	13. Mar 2008	ttu	Completely reviewed
5.03	18. Mar 2008	ttu	Formatting changes, corrections

The information furnished in this document is believed to be accurate and reliable. However, no responsibility is assumed by sevenstax for its use, nor for any infringements of patents or other rights of third parties resulting from its use. No license is granted under any patents or patent rights of sevenstax.

This document is an intellectual property of sevenstax GmbH. Unauthorized copying and distribution is prohibited.

Copyright (c) 2002-2008 by sevenstax GmbH